İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

IMPLEMENTATION OF CHAT AND WHITEBOARD APPLICATION USING REMOTE METHOD INVOCATION (JAVA, RMI)

M.Sc. Thesis by

A. Çağatay TUNALI, B. S.

Department: COMPUTER ENGINEERING

Programme: Computer Engineering

JANUARY 2003

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

IMPLEMENTATION OF CHAT AND WHITEBOARD APPLICATION USING REMOTE METHOD INVOCATION (JAVA, RMI)

M.Sc. Thesis by

A. Çağatay TUNALI, B.S.

(504001557)

Date of submission : 24 December 2002

Date of defence examination: 15 January 2003

Supervisor (Chairman): Assoc. Prof. Dr. Coşkun SÖNMEZ

Members of the Examining Committee Prof.Dr. Bülent ÖRENCİK (ITU.)

Dr. Erdal ÇAYIRCI (Turkish War Colleges)

JANUARY 2003

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

UZAKTAN METOT İSTEME İLE SOHBET VE BEYAZTAHTA İSTEMCİ VE SUNUCUNUN TANIMLANMASI VE YAPILANDIRILMASI

YÜKSEK LİSANS TEZİ

Müh. A. Çağatay TUNALI

(504001557)

Tezin Enstitüye Verildiği Tarih : 24 Aralık 2002

Tezin Savunulduğu Tarih : 15 Ocak 2003

Tez Danışmanı : Doç.Dr. Coşkun SÖNMEZ
Diğer Jüri Üyeleri Prof.Dr. Bülent ÖRENCİK (İ.T.Ü.)

Dr. Erdal ÇAYIRCI (Harp Akademileri)

OCAK 2003

PREFACE

I would like to thank my instructor, Assoc. Prof. Dr. Coşkun SÖNMEZ who diligently scrutinized my thesis and made many valuable comments and suggestions.

I would also like to thank Garanti Technology and its deputy general manager Rüştü KARACA for their assistance and support.

Thanks to my valued professional colleague: Tacettin AYAR for supplying CORBA related documents and information.

Finally, thanks to all my family for their understanding and support for all my educational life.

January, 2002 Ahmet Çağatay TUNALI

TABLE OF CONTENTS

ABBREVIATION

ACL : Access Control List
API : Application Programming Interface
AWT : Abstract Windowing Toolkit
BOA : Basic Object Adapter
CDR : Common Data Representation
CGI : Common Gateway Interface
CLSID : Class Identifier
CORBA : Common Object Request Broker Architecture
DCOM : Distributed Common Object Model
DII : Dynamic Invocation Interface
DSI : Dynamic Skeleton Interface
GIOP : General inter-ORB Protocol
GUI : Graphical User Interface
HTTP : Hypertext Transfer Protocol
I/O : Input/Output
ID : Identification
IDL : Interface Definition Language
IIOP : Internet Inter-ORB Protocol
IOR : Interoperable Object Reference
IPC : Inter Process Communication
IP : Internet Protocol
IR : Interface Repository
ISO : International Standards Organization
ITU : International Telecommunications Union
JAR : Java Archive
JNDI : Java Naming and Directory Interface
JRMP : Java Remote Method Protocol
JSR : Jump Subroutine Instruction
JVM : Java Virtual Machine
LAN : Local Area Network
OA : Object Adapter
OAD : Object Activation Daemon
OMG : Object Management Group
ORB : Object Request Broker
OSI : Open System Interconnection
PC : Personal Computer
POA : Portable Object Adaptor
RPC : Remote Procedure Call
RMI : Remote Method Invocation
SMP : Symmetric Multiprocessing
SII : Static Invocation Interface
TCP : Transmission Control Protocol
UDP : User Datagram Protocol
URL : Uniform Resource Locator
WAN : Wide Area Network

TABLE LIST

FIGURE LIST

ÖZET

Nesne tabanlı teknolojiler uzun yıllardır tek kullanıcılı sistemlere odaklanmıştır. Uygulumalar büyüdükçe ve karmaşıklaştıkça istemci/sunucu teknolojileri ortaya çıkmaya başladı. Bu da çoklu-kullanıcılı platformlarda paylaşılan nesne teknolojisine olan ihtiyacı doğurdu. Bu probleme bir çözüm de; dağıtılmış nesne teknolojisiydi. Dağıtılmış nesneler, birden fazla bilgisayarın bulunduğu ağlarda kullanıcılar arasında etkileşimli iletişimi sağladılar. Bu mimari ayrıca, hesap yükünün bilgisayarlar arasında dağılımını ve bilgi değiş tokuşunu gerçekleştirdi. Ağ programlamayı kolaylaştırmak ve yeniden kullanımı sağlamak için üç standart ortaya çıktı. Bunlardan birincisi OMG (Nesne Yönetim Grubu) tarafından yaratılan CORBA (Genel Nesne İstek İstemci Mimarisi), ikincisi Microsoft tarafından yaratılan DCOM (Dağıtılmış Bileşen Nesne Modeli), son olarak Java Microsystems tarafından yaratılan RMI (Uzaktan Metod Çağırma) oldu. Bu teknolojiler arasından doğru tercihi yapmak için, teknolojiler kapsamlı olarak karşılaştırılmışlardır.

Sonuç olarak, CORBA ile DCOM teknolojilerinin aşağı yukarı eşit oldukları görülmüştür. RMI'ın bunlara göre performansının düşük olduğu kaydedilmiştir. CORBA mimarisi geniş tabanlı daha önceden oluşturulmuş sistemleri birleştirebilecek bir teknolojidir. DCom mimarisi ise aynı işlemi sadece Windows platformuna bağımlı olarak gerçekleştirmektedir. RMI ise platformdan bağımsız olarak, yeni gerçekleştirilecek olan dağıtılmış uygulamalar için; bu teknolojiler arasında en iyi seçimdir. Bu yargıya varmamızın nedeni, RMI sisteminin kolay ve rahat uygulanır olması, ve sistemin tamamıyla nesne tabanlı olmasından ileri gelir. Gelecekte bu teknolojilerin varlıklarını sürdüreceklerine ve daha fazla işbirliği yapacaklarına inanmaktayız.

SUMMARY

Object oriented technology was focused on single-user environment for many years. As applications grew to become more complex and client/server technology emerged, there was a need to have shared objects in multi-users environment. One solution is the use of distributed objects, where objects executing in multiple computers interact over the network to participate in application processes. This architecture allows the workload to be distributed and it also allows independently developed solutions implemented in different environment and platform to interact with each other. To simplify network programming and to realize component-based software architecture, three distributed object models have emerged as standards, Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM), and Sun Microsystems's Remote Method Invocation (RMI). In order to make the right choice between these technologies, technologies were thoroughly compared. The ease of deployment was also considered.

The conclusions are that the performance between CORBA and DCOM is almost equivalent. RMI is little slower than these two technologies. CORBA is the dominant remote architecture, connecting large-scale enterprise systems, which demands integration with legacy systems. DCOM is part of COM+, which is the dominant component architecture, operating mainly on the Windows platforms. RMI is a platform independent technology that makes practice of distributing applications more convenient and easier than the others. For the new distributed applications RMI is the best choice. We believe that in the future, these technologies will coexist and cooperate.

1. INTRODUCTION

In recent days, we can observe rapid development of the Internet. Infrastructure of this global network grows and every year new services are available. This unprecedented growth was caused by the introduction of the WWW technologies that are still driving forces of the Internet expansion [18]. The popularity and the ubiquity of the Internet make it ideal basis for construction of the collaborative environments. In fact, the creation of a collaboration environment for the geographically dispersed users was one of the reasons for creation of this network of networks. Collaborative tools such as the electronic mail have been present in this environment for a long time. However, recently many groups of the Internet users declared need for more sophisticated collaborative tools.

As the features that is supplied by the tool increases the complexity of the programming increases. The sophisticated needs necessitate the usage of the new technology. There are three methods that will help us to implement the collaborative system. These are

- Socket Programming
- Implementing application specific distributed system
- Using the standard distributed system

In order to make decision on methods, first we must decide the server and client architecture. After selecting architecture of the system, we can decide the implementation method. Each implementation method has its advantages and disadvantages. Socket programming is the one of the method that will be chosen. But this method needs more programming work and time then the other methods. Another disadvantage of socket programming is we cannot add new features or new applications easily in the system. We must know all the system functionality and on the other hand, there will be some cases that require lots of changes of all system. Second method is nearly as same as the third method. But this method needs more programming work and time then the other two methods. Another disadvantage is

the distributed system applications are already implemented in computer's world. We can use the existing ones. So the third method seems more reasonable then the others.

As an application, a shared chat and whiteboard system is implemented on RMI system. This system is chosen for its simplicity and suitability to other systems. More details are given in fourth chapter.

In the first chapter, the anatomy of the distributed application is discussed. After this discussion, requirements of this kind of systems are given. After giving the system requirements, a programming language which is used in selected distributed system is discussed on topic "how it meets the requirements". And finally, a new technology distributed objects is described in this chapter.

In the second part, distributed object system technologies are discussed. First an old technology, "RPC", is described. After that CORBA and COM technologies are discussed. CORBA is a distributed system that provides a mechanism to use functions, objects between two or more programming environment. COM is also a distributed system that enables the functionality of using different programming environment objects in Microsoft operating systems. Finally, RMI (Remote Method Invocation) which is selected as a development system in the project is presented. This system is purely defined for and by Java.

In the third part, collaborative systems and their problems and needs are presented. This part of the document tries to identify typical problems of collaborative systems. It also tries to identify the solutions to those problems and show how they are employed in the existing environments. Also some existing collaborative systems are compared according to their structure and functionality.

In the fourth part, the infrastructure of shared chat and whiteboard system is presented. Also, the document describes how the collaboration system problems are solved in this system. The client and server implementations, the layers for communication between server, client and database are shown.

2. DISTRIBUTED SYSTEMS

In the 1970's, networking began to emerge as an important aspect of computer systems. Driven by applications in the military and airline industries, computer systems were connected and inter-operation became widespread. During the 1980's, distributed computing became a vital aspect of many computer systems. In the early 2000's, we are beginning to see the emergence of ubiquitous computing: characterized as a massive heterogeneous "sea" of disparate computational devices, with varying connection bandwidths and an ever-changing topology of connections.

Before the invention of computers, processing information was both slow and tedious. The advent of computers has transformed the world, and the way in which we work with information. However, using and storing this information in isolation, like any expensive resource is inefficient. Ergo, unless our computers are to exist in isolation, we require methods that allow computers to meaningfully interact, and ways of transferring information between them. Communication networks, which interconnect computers and allow them to work in concert, are a common solution to this problem.

However, merely physically connecting computers is not enough to achieve logical interaction in its own right. Computers must adhere to a common set of rules or protocols for defining their interactions. By connecting separate computers, we make it possible for the programs executing on those computers to interact. When processes on separate computers interact, we term the whole a distributed system.

2.1 Distributed Communication

Distributed computing as we understand it today is a far cry from the limited facilities of early distributed systems, such as remote job entry handlers. Their role however was simple - to allow scarce and expensive information and resources to be shared by users. Ever since computer users began accessing central processor resources

from remote terminals over 40 years ago, computer networks have become more versatile, more powerful and inevitably more complex.

At the heart of distributed computing are communication networks. They are the infrastructures that support information flow between computers. The initial development of such networks was fostered through experimental networks such as ARPANET and CYCLADES. ARPANET, which went live in December 1969, was initially motivated by the requirements of the US Military for a communications network that could survive a nuclear war. This early work established the procedures for connecting computers and facilitating their interaction. Just physically connecting computers was not sufficient to ensure successful interaction though. Two computers wishing to communicate must adhere to a common set of rules for defining their interactions. This rule set is termed a protocol, and is an agreement between the communicating parties on how communication is to proceed [2].

To reduce their design complexity, network architectures are organized as a series of layers or levels of abstraction, each built upon the preceding one. Whilst the number and nature of these layers may differ between architectures, their purpose is similar: to offer services to the higher layers, shielding them from the details of how the offered services are actually implemented. Each layer has its own particular communication protocol, and collections of protocols defined in terms of a common framework are known as a protocol suite or stack.

In early computer systems, it was common for each application found on a computer to employ its own protocol stack. This communication support was usually built into the application, and was not available for use by any other applications. This approach therefore had the inherent disadvantages of duplicated functionality and inefficient resource usage. To alleviate this undesirable situation, research focused on providing communication mechanisms at the operating system level through the provision of shared communication suites. Although a vast improvement, facilities provided by the operating system were invariably specific to the particular type of computer on which they were executing. In the mid 1970s, computer vendors began to develop their own network architectures, to enable communication between their own ranges of machines. Important examples of this period are the Internet model that emerged from ARPANET, IBM's Systems Network Architecture (SNA) and Digital's DECnet. This meant however, that since each suite was developed for the vendors' own machines, they were usually composed of proprietary (closed) protocols [3]. This situation posed two considerable problems:

- Systems from competing vendors were not able to interoperate
- The communication specification was controlled by a single organization

Since the vendors controlled the protocol specification, they also had the power to change the specification at their discretion. Understandably, this made third party developers very nervous in adopting and working to a standard whose specification might be changed at any given moment. Although subsequent publishing of the protocol specifications aided their widespread adoption, the issue remained. Further, as each proprietary communication suite evolved, systems from competing manufacturers became even more incompatible.

The splintered evolution of incompatible communication suites forced the computing community to realize that standards were required to enable interaction between different types of computer. In 1977, the International Standards Organization (ISO) began working towards defining a non-proprietary (open) suite of protocols. The resulting standard is known as the ISO Open Systems Interconnection (OSI) reference model, and is jointly defined by ISO and the International Telecommunications Union (ITU-T). Most of the proprietary suites that preceded the OSI model have since undergone modification and are now considered as specialized incarnations of the OSI model.

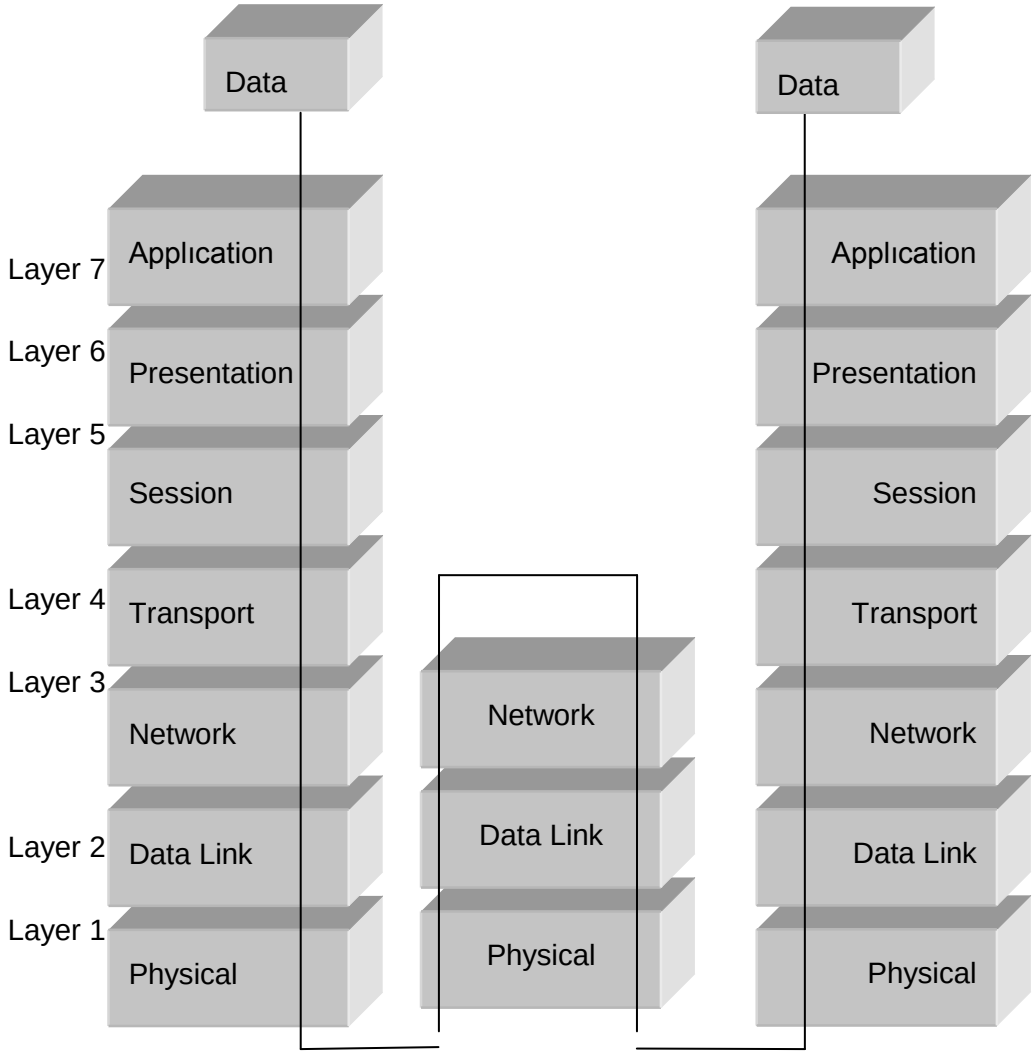

Figure 1.1 - The OSI reference model

The OSI Reference model is structured into seven layers that represent the logical sequence of functions carried out when messages are constructed for transmission, dispatched, and then dismantled on arrival. It also serves to provide a common basis for the co-ordination of communication systems standards development and to allow existing standards to be placed into perspective. An example of the OSI Reference Model is shown in figure 1.1 [3]. Data at Host A is translated by the OSI stack into a form that can be communicated over the wire. It is then sent over the wire (perhaps via some network nodes), before it is reconstituted at Host B by the corresponding protocol suite, before finally being made available to the destination application.

Of particular interest to this thesis is Layer 7 – the Application layer. The Application layer is the highest level of abstraction defined in the OSI model and is ultimately responsible for managing the communications between applications. It provides

programming primitives that a developer is able to use to access the communication facilities offered by the full protocol suite.

2.2 Anatomy of Distributed Applications

A distributed application is built upon several layers. At the lowest level, a network connects a group of host computers together so that they can talk to each other. Network protocols like TCP/IP let the computers send data to each other over the network by providing the ability to package and address data for delivery to another machine. Higher-level service can be defined on top of the network protocol, such as directory services and security protocols. Finally, the distributed application itself runs on top of these layers, using the mid-level services and network protocols as well as the computer operating systems to perform coordinated tasks across the network [2].

At the application level, a distributed application can be broken down into the following parts:

2.2.1 Process

A typical computer operating system on a computer host can run several processes at once. A process is created by describing a sequence of steps in a programming language, compiling the program into an executable form, and running the executables in the operating system. While it's running, a process has access to the resources of the computer through the operating system. A process can be completely devoted to particular applications, or several applications can use a single process to perform tasks.

2.2.2 Threads

Every process has at least one thread of control. Some operating systems support the creation of multiple threads of control within a single process. Each thread in a process can run independently from the other threads, although there is usually some synchronization between them. One thread might monitor input from a socket connection, while another might listen for user's events and provide feedback to the user through output devices. At some point, input from the input stream may require feedback from the user. At this point, the two threads will need to coordinate the transfer of input data to the user's attention.

2.2.3 Objects

Programs written in object-oriented languages are made up of cooperating objects. One simple definition of an object is group of related data, with methods available for querying or altering the data, or taking some action based on the data. A process can be made up of one or more objects, and these objects can be accessed by one or more threads within the process. And with the introduction of distributed object technology like RMI and CORBA, an object can also be logically spread across multiple processes, on multiple computers.

2.2.4 Agents

Agent is a computing entity that is a bit more intelligent and autonomous than an object. An agent is supposed to be capable of having goals that it needs to accomplish. Some agents can monitor their progress towards achieving their goals at a higher level that just successful execution methods, like an object.

Agents can be distributed across multiple processes, and can be made up of multiple objects and threads in these processes.

Distributed application can be thought of as a coordinated group of agent working to accomplish some goal. Each of these agents can be distributed across multiple processes on remote hosts, and consist of multiple objects or threads of control.

2.3 Requirements for Developing Distributed Applications

2.3.1 Partitioning and Distributing Data and Functions

Computational tasks can be distributed based on the data needs of the applications: maximize local data needed for processing, and minimize data transfers over the network. In other, more compute-intensive applications, you can partitions the system based upon the functional requirements of the system, with the data mapped to the most logical compute host. This method of partitioning is especially useful when the overhead associated with the data transfers is negligible compared to the computing time spent at the various hosts [2].

In the best of all possible worlds, you can develop modules based upon either data- or functionally driven partitioning. You could then distribute these modules as

needed throughout a virtual machine comprised of computers and communication links, and easily connect the modules to establish the data flow required by the application. These module interconnections should be as flexible and transparent as possible, since they may need to be adjusted at any point during development or deployment of the distributed system.

2.3.2 Flexible, Extendible Communication Protocols

The type and format of the information that's sent between agents in distributed systems is subject to many varied and changing requirements. Some of them are a result of the data/function partitioning issues. The allocation of tasks and data to agents in the distributed system has a direct influence on what type of data will need to be communicated between agents, how much data will be transferred , and how complicated the communication protocol between agents needs to be. If most of our data is sitting on the host where it's needed, then communications will be mostly short, simple messages to report status, instruct other agents to start processing. If central data servers are providing lots of data to remote agents, then the communication protocol will be more complex and connections between nodes in the system will stay open longer.

The communication protocols a given agent will need to understand might also be dictated by legacy systems that need to be incorporated into the system. These legacy systems might control data or functionality that's critical to enabling a given system, but are not easily transferable to a new system. Support for these protocols should be available, or easily attainable, in distributed application development environment. In the worst case, when support for required protocol is unavailable due to its obscurity or the expense associated with the available support, the required protocol can be developed or incorporate the existing infrastructure with the extended communication abilities.

2.3.3 Multithreading Requirements

Agents often have to execute several threads of control at once, either to service requests from multiple remote agents, or block I/O while processing data, or for any number of other reasons. Multithreading is often an effective way to optimize the use of various resources. The ability to create and control multiple threads of control is especially important in developing distributed applications, since distributed agents are typically more asynchronous than agents within a single process on a single

host. The environments in which agents are running can be very heterogeneous, and every agent in a distributed application must not be slave to the slowest, most heavily loaded agent in the system.

2.3.4 Security

The information transactions that occur between computing agents often need to be secure from outside observation, when information of sensitive nature needs to be shared between agents. In situations where an outside agent not under the host's direct control is allowed to interact with local agents, it is also wise to have reasonable security measures available to authenticate the source of the agent, and to prevent the agent from wreaking havoc once it gains access to local processing resources. So at a minimum a secure distributed application needs a way to authenticate the identity of agents, define resource access level for agents, and encrypt data for transmission between agents.

2.4 What Does Java Provide?

The original design motivations behind Java and its predecessor, Oak, were concerned mainly with reliability, simplicity, and architecture neutrality. Subsequently, as the potential for Java as an "Internet Programming Language" was seen by its developers at Sun Microsystems, support for networking, security, and multithreaded operations was incorporated or improved. All of these features of the Java language and environment also make for a very powerful distributed application development environments [2].

2.4.1 Object-Oriented Environment

Java is a "pure" object-oriented language; in the sense that smallest programmatic building block is a class. A data structure or function cannot exist or be accessed at runtime except as an element of a class definition. This results in a well-defined, structured programming environment in which all domain concepts and operations are mapped into class representation and transactions between them. This is advantageous for systems development in, but also has benefits specifically for you as the distributed system developer. An object, as an instance of a class, can be thought of as a computing agent. Its level of sophistication as an autonomous agent is determined by the complexity of its methods and data representations, as well as

its role within the object model of the system, and the runtime object community defining the distributed system. Distributing a system implemented in Java, therefore, can be thought of as simply distributing its objects in a reasonable way, and establishing networked communication links between them using Java's built-in network support.

2.4.2 Abstract Interfaces

Java's support for abstract object interface is another valuable tool for developing distributed systems. An interface describes the operations, messages, and queries a class of objects is capable of servicing, without providing any information about how these abilities are implemented. If a class is declared as implementing a specified interface, then the class has to implement the methods specified in the interface. The advantages of implementation-neutral interfaces are that other agents in the system can be implemented to talk to the specified interface without knowing how the interface is actually implemented in a class. By insulating the class implementation from those using the interface, we can change the implementation as needed. If a class needs to be moved to a remote host, then the local implementation of the interface can act as a surrogate or stub, forwarding calls to the interface over the network to the remote class.

Abstract interfaces are powerful part of the Java language and are used to implement critical elements of the Java API. The platform independence of the Abstract Windowing Toolkit (AWT) is accomplished using abstract component interfaces that are implemented on each platform using the native windowing system.

2.4.3 Platform Independence

Code written in Java can be compiled into platform-independent bytecodes using Sun's Java compiler, or any of the many third-party Java compilers now on the market. These bytecodes run on the Java Virtual Machine, a virtual hardware architecture which is implemented in software running on a "real" machine and its operating system. Java bytecodes can be run on any platform with a Java Virtual Machine.

This allows virtually any available PC or workstation to be home to an agent in a distributed system once the elements of the system have been specified using Java

classes and compiled into Java bytecodes, they can migrate without recompilation to any of the hosts available. This makes for easy data-and load-balancing across network. There is even support in the Java API for downloading a class definition through a network connection, creating an instance of the class, and incorporating the new object into the running process. This is possible because Java bytecodes are runnable on the Java Virtual Machine, which is guaranteed to be underneath any Java application.

2.4.4 Fault Tolerance through Exception Handling

Java supports throwing and catching errors and exceptions, both system-defined and application-defined. Any method can throw an exception; it is the calling method's responsibility to handle the exception, or propagate the exception up the calling chain. Handling an exception is a matter of wrapping any potential exception-causing code with a try/catch/finally statement, where each catch clause handles a particular type of exception. If a method chooses to ignore particular exceptions, then it must declare that it throws the exception it is ignoring. When a called method generates an exception, it will be propagated up the calling chain to be handled by a catch clause in a calling method, or, if not, to result in a stack dump and exit from the Java process. After all is said and done, whether the try block runs to completion without a problem, or an exception gets thrown, the code in the finally block is always called. So you can use the finally block to clean up any resource you created in the try block, for example, and be sure that the cleanup will take place whether an exception is thrown or not.

An agent can be written to handle the exceptions that can be thrown by each method it calls. Additionally, since any subclass of java.io.Throwable can be declared in a method's thrown clause, an application can define its own types of exceptions to indicate specific abnormalities. Since an exception is represented as an object In the Java environment, these application-specific exceptions can carry with them data and methods that can be used to characterize, diagnose, and potentially recover form them.

2.4.5 Network Support

The Java API includes multilevel support for network communications. Low-level sockets can be established between agents, and data communication protocols can be layered on top of the socket connection. APIs built on top of the basic networking

support in Java provide higher-level networking capabilities, such as distributed objects, remote connections to database servers, directory services, etc.

Java's network support provides a quick way to develop the communication elements of a basic distributed system. Java's other core features, such as platform-independent bytecodes, facilitate the development of more complex network transactions, such as agents dynamically building a protocol for talking to each other by exchanging class definitions. The core Java API also includes built-in support for sharing Java objects between remote agents.

2.4.6 Security

Java provides two dimensions of security for distributed systems? A secure local runtime environment and the ability to engage in secure remote transactions.

2.4.6.1 Runtime Environment

At the same time that Java facilitates the distribution of system elements across the network, it makes it easy for the recipient of these system elements to verify that they cannot compromise the security of the local environment. If a Java code is run in the context of an applet, then the Java Virtual Machine places rather severe restrictions on its operation and capabilities. It's allowed virtually no access to the local file system, very restricted network access, no access to local code or libraries outside of the Java environment, and restricted thread manipulation capabilities, among other thing. In addition, any class definitions loaded over the network are subjected to a stringent bytecode verification process, in which the syntax and operations of the bytecodes are checked for incorrect or potentially malicious behavior.

2.4.6.2 Secure Remote Transactions

This part allows the programmers to use authentication, access level control, and encryption in their Java applications. This capability of the environment makes it easy to add user authentication and data encryption to establish secure network links, assuming that the basic encryption and authentication algorithms already exists.

2.4.7 Multithreading Support

The ability to generate multithreaded agents is a fundamental feature of Java. Any class that you create can extend the java.lang.Thread class by providing its own implementation of a run method. When the thread is started, this run method will called and class can do its work within a separate thread of control. This is one way to delegate tasks to threads in a given agent; another is to have your workhorse classes derive from the java.lang.Runnable, and allocate them as needed to threads or thread groups. Any class implementing the runnable interface can be wrapped with a thread by simply creating a new thread with the runnable object as the argument.

Java also lets the performance of a given agent tweaked through control and manipulation of its thread. Threads are assigned priorities that are publicly pollable and settable, giving the ability to suggest how processing time is allocated to threads by the Virtual Machine. Threads can also be made to yield to other threads, to sleep for some period of time, to suspend indefinitely, or to go way altogether.

2.5 Distributing Objects

Distributed objects are a potentially powerful tool that has only become broadly available for developers at large in the past few years. The power of distributing objects is not in the fact that a bunch of objects are scattered across the network. The power lies in that any agent in your system can directly interact with an object that "lives" on a remote host. Distributed objects, if they're done right, really give you a tool for opening up your distributed system's resources across the board. And with a good distributed object scheme you can do this as precisely or as broadly as you'd like [2].

2.5.1 Why Distribute Objects?

Distributed object systems try to address these issues by letting developers take their programming objects and have them "run" on a remote host rather than the local host. The goal of most distributed object systems is to let any object reside anywhere on the network, and allow an application to interact with these objects exactly the same way as they do with a local object. Additional features found in some distributed object schemes are the ability to construct an object on one host

and transmit it to another host and the ability for an agent on one host to create a new object on another host.

The value of distributed objects is more obvious in larger, more complicated applications than in smaller, simpler ones. That's because much of the trade-off between distributed objects and other techniques, like message passing, is between simplicity and robustness. In a smaller application with just a few object types and critical operations, it's not difficult to put together a catalog of simple messages that would let remote agents perform all of their critical operation through on-line transactions. With a larger application, this catalog of messages gets complicated and difficult to maintain. It's also more difficult to extend a large message-passing system if new objects and operations are added. So being able to distribute the objects in our system directly saves us a lot of design overhead, and makes a large distributed system easier to maintain in the long run.

2.5.2 Creating Remote Objects

The essential requirements in a distributed object system are the ability to create or invoke objects on a remote host or process, and interact with them as if they were objects within our own process. It seems logical that we would need some kind of message protocol for sending requests to remote agents to create new objects, to invoke methods on these objects, and to delete the objects when we're done with them. The networking support in the Java API makes it very easy to implement a message protocol.

To create a remote object, we need to reference a class, provide constructor arguments for the class, and receive a reference to the created object in return. This object reference will be used to invoke methods on the object, and eventually to ask the remote agent to destroy the object when we are done with it. So the data we will need to send over the network include class references, object references, method references, and method arguments.

ClassLoader can be used to send class definitions over the network. If we want to create a new remote object from a given class, we can send the class definition to the remote host, and tell it to build one using a default constructor. Object references require some thought, though. These are not the same as local Java object references. We need to have an object reference that we can package up and send over the network, i.e., one that's serializable. This object reference, once

we receive it, will still need to refer back to the original object on the remote host, so that when we call methods on it the method invocations are deferred to the "source" object on the remote host. One simple way to implement remote object references is to build an object lookup table into the remote agent. When a client requests a new object, the remote agent builds the requested object, puts the object into the table, and sends the table index of the object to the client. If we use sockets and streams to send requests and object references back and forth between remote agents, a client might request a remote object with something like this:

2.5.3 Remote Method Calls

Now that the requestor has a reference to an object on the remote host, it needs a way to invoke methods on the object. Since Java allows us to query a class or object for its declared methods and data members, the local agent can get a direct reference to the method that it wants to invoke on the remote object, in the form of a method object.

In distributed object systems, the tasks of packaging up method arguments for delivery to the remote object, and the task of gathering up method return values for the client, are referred to as data marshaling. One approach we can take to data marshaling is to turn every object argument in a remote method call into a remote object just like we did previously, by generating an object reference and sending that to the remote agent as the method argument. If the method returns an object value, then the remote host can generate a new object reference and send that back to the local host.

2.5.4 Features of Distributed Object Systems

An object interface specification is used to generate a server implementation of a class of objects, an interface between the object implementation and the object manager, sometimes called an object skeleton, and a client interface for the class of objects, sometimes called an object stub. The skeleton will be used by the server to create new instances of the class of objects and to route remote method calls to the object implementation. The stub will be used by the client to route transactions (method invocations, mostly) to the object on the server. On the server side, the class implementation is passed through a registration service, which registers the new class with a naming service and an object manager, and then stores the class in the server's storage for object skeletons.

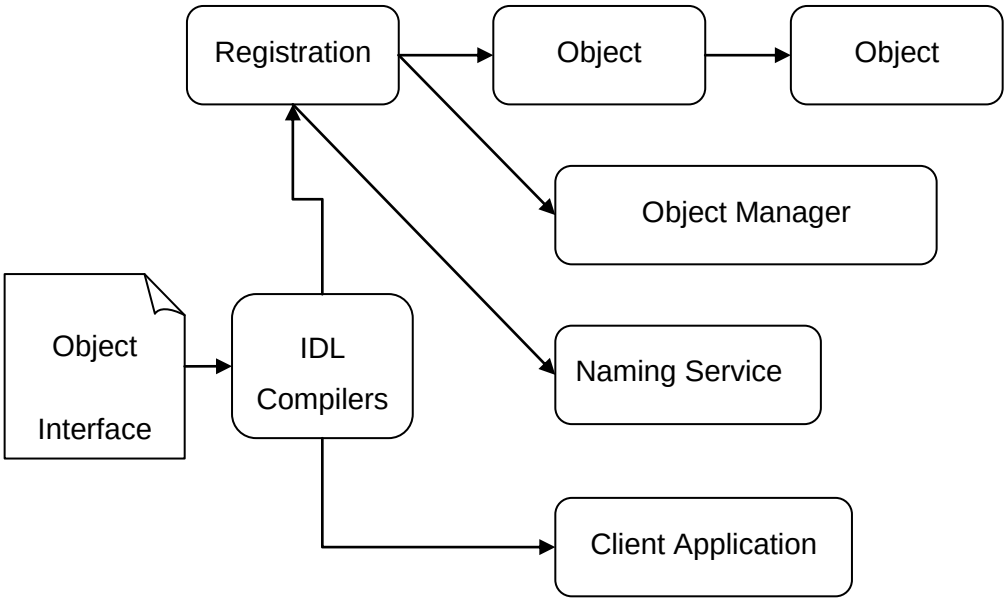

Figure 1.2 - General architecture for distributed object systems

With an object fully registered with a server, the client can now request an instance of the class through the naming service. The runtime transactions involved in requesting and using a remote object are shown in figure 1.2 [2]. The naming service routes the client's request to the server's object manager, which creates and initializes the new object using the stored object skeleton. The new object is stored in the server's object storage area, and an object handle is issued back to the client in the form of an object stub interface. This stub is used by the client to interact with the remote object.

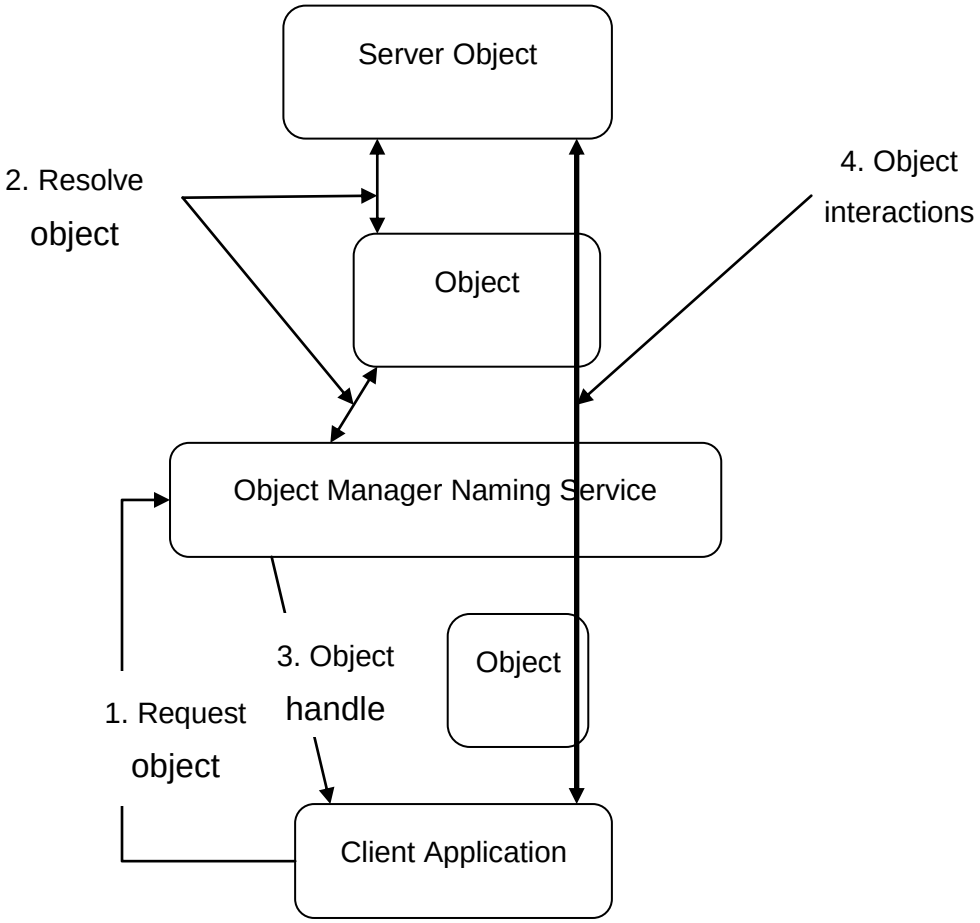

Figure 1.3 - Remote object transactions at runtime

While figure 1.3 illustrates a client-server remote object environment, a remote object scheme can typically be used in a peer-to-peer manner as well. Any agent in the system can act as both a server and a client of remote objects, with each maintaining its own object manager, object skeleton storage, and object instance storage. In some systems the naming service can be shared between distributed agents, while in others each agent maintains its own naming service [2].

2.5.5 Object Interface Specification

To provide a truly open system for distributing objects, the distributed object system should allow the client to access objects regardless of their implementation details, like hardware platform and software language. It should also allow the object server to implement an object in whatever way it needs to.

Some distributed object systems provide a platform-independent means for specifying object interfaces. These object interface descriptions can be converted

into server skeletons, which can be compiled and implemented in whatever form the server requires. The same object interfaces can be used to generate client-side stub interfaces. If we're dealing with a Java-based distributed system, then the server skeletons and client stubs will be generated as Java class definitions, which will then be compiled into bytecodes.

2.5.6 Object Manager

The object manager is really at the heart of the distributed object system, since it manages the object skeletons and object references on an object server. When a client asks for a new object, the object manager locates the skeleton for the class of object requested, creates a new object based on the skeleton, stores the new object in object storage, and sends a reference to the object back to the client. Remote method calls made by the client are routed through the manager to the proper object on the server, and the manager also routes the results back to the client. Finally, when the client is through with the remote object, it can issue a request to the object manager to destroy the object. The manager removes the object from the server's storage and frees up any resources the object is using.

Some distributed object systems support things like dynamic object activation and deactivation, and persistent objects. The object manager typically handles these functions for the object server. In order to support dynamic object activation and deactivation, the object manager needs to have activation and deactivation method registered for each object implementation it manages. When a client requests activation of a new instance of an interface, for example, the object manager invokes the activation method for the implementation of the interface, which should generate a new instance. A reference to the new instance is returned to the client. A similar process is used for deactivating objects. If an object is set to be persistent, then the object manager needs a method for storing the object's state when it is deactivated, and for restoring it the next time a client asks for the object.

Depending on the architecture of the distributed object system, the object manager might be located on the host serving the objects, or its functions might be distributed between the client and the server, or it might reside completely on a third host, acting as a liaison between the object client and the object server.

2.5.7 Registration/Naming Service

The registration/naming service acts as an intermediary between the object client and the object manager. Once we have defined an interface to an object, an implementation of the interface needs to be registered with the service so that it can be addressed by clients. In order to create and use an object from a remote host, a client needs a naming service so that it can specify things like the type of object it needs, or the name of a particular object if one already exists on the server. The naming service routes these requests to the proper object server. Once the client has an object reference, the naming service might also be used to route method invocations to their targets.

If the object manager also supports dynamic object activation and persistent objects, then the naming service can also be used to support these functions. If a client asks the service to activate a new instance of a given interface, the naming service can route this request to an object server that has an implementation of that interface. And if an object manager has any persistent objects under its control, the naming service can be notified of this so that requests for the object can be routed correctly.

2.5.8 Object Communication Protocol

In order for the client to interact with the remote object, a general protocol for handling remote object requests is needed. This protocol needs to support, at a minimum, a means for transmitting and receiving object references, method references, and data in the form of objects or basic data types. Ideally we don't want the client application to need to know any details about this protocol. It should simply interact with local object interfaces, letting the object distribution scheme take care of communicating with the remote object behind the scenes. This minimizes the impact on the client application source code, and helps you to be flexible about how clients access remote services.

2.5.9 Development Tools

We'll need to develop, debug, and maintain the object interfaces, as well as the language-specific implementations of these interfaces, which make up our distributed object system. Object interface editors and project managers, language cross-compilers, and symbolic debuggers are essential tools. The fact that we are developing distributed systems imposes further requirements on these tools, since

we need a reasonable method to monitor and diagnose object systems spread across the network. Load simulation and testing tools become very handy here as well, to verify that our server and the network can handle the typical request frequencies and types we expect to see at runtime.

2.5.10 Security

Any network interactions carry the potential need for security. In the case of distributed object systems, agents making requests of the object broker may need to be authenticated and authorized to access elements of the object repository, and restricted from other areas and objects. Transactions between agents and the remote objects they are invoking may need to be encrypted to prevent eavesdropping. Ideally, the object distribution scheme will include direct support for these operations. For example, the client may want to "tunnel" the object communication protocol through a secure protocol layer, with public key encryption on either end of the transmission.

3. DISTRIBUTED SYSTEM TECHNOLOGIES

With the wide spread utilization of object technology, it has become more and more important to employ the object oriented paradigm in distributed environments as well. This raises several inherent issues, such as references spanning address spaces, the need to bridge heterogeneous architectures, etc. It is the main goal of this section to provide an architectural analysis of current software platforms in this area. We focus on the following key distributed object platforms: CORBA, COM/DCOM, and Java RMI. The first one, CORBA, is specified by OMG, which is the largest consortium in the software industry. CORBA has undergone an evolution ranging from CORBA 1.0 (1991) and CORBA 2.0 (1995) to CORBA 3.0, which is soon to be released. The second platform analyzed is the Microsoft Component Object Model (COM). This platform has also been evolving gradually along the milestones OLE, COM, DCOM, and COM +. The Java environment, designed by Sun Microsystems, has probably experienced the greatest evolution recently. From the broad spectrum of the Java platform segments, we will focus on Java RMI, which targets working with distributed objects [10, 11].

RPC system is also a distributed system which is implemented before the object oriented programming. Although the system is not implemented with an object oriented language, it is a base for all distributed systems. In order to get the idea we will first focus on the old technology RPC.

Remote Procedure Calls (RPC) is a mechanism that facilitates a request/reply interaction between two distributed processes. This is similar to the traditional mechanism of procedure calls found in high-level programming languages. The fundamental difference is that the calling procedure executes in one computing machine, and the called procedure executes in another, whilst data is exchanged between the two communicating parties. The mechanism for RPC was modeled directly on the inter process communication (IPC) facilities. Indeed, so successful were they that RPC has no distinction in syntax between a local and a remote procedure call [4].

During an RPC call there are five separate modules that interact to enable the call. They are the client, the client-stub, the RPC communications package (RPC Runtime), the server-skeleton and the server. When the client wishes to call a procedure that exists on a remote machine, it invokes the appropriate method in the client-stub. To the client, this resembles a normal local procedure call. The client-stub then assembles one or more data packets that include the target procedure and the required arguments. These packets are then passed to the local RPC Runtime, which transmits them to the remote Runtime. On receipt, these packages are passed to the server-skeleton, where they are unpacked and passed to the target procedure in the server. Once this procedure has been executed, any results are packaged up and the process repeated in reverse. RPC is synchronous in nature, so while the server procedure is executing, the client is suspended, awaiting the result. The RPC Runtime (or request broker) establishes a client/server relationship between the interacting parties, removing the need for each party to be aware of the other's location. The figure 2.1 shows the system architecture.

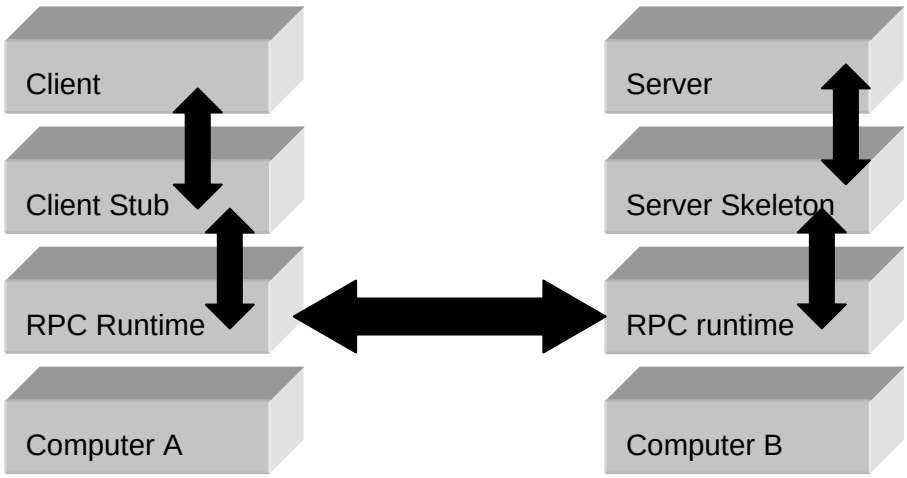

Figure 2.1 - A remote procedure call

3.1 An Overview of Remote Procedure Call (RPC) System

3.1.1 The RPC Model

The Remote Procedure Call (RPC) protocol is based on the remote procedure call model, which is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location (such as a register

window). It then transfers control to the procedure, and eventually regains control. At that point, the results of the procedure are extracted from the well-specified location, and the caller continues execution [24].

The remote procedure call model is similar. One thread of control logically winds through two processes: the caller's process, and a server's process. The caller process first sends a call message to the server process and waits (blocks) for a reply message. The call message includes the procedure's parameters, and the reply message includes the procedure's results. Once the reply message is received, the results of the procedure are extracted, and caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message.

In this model, only one of the two processes is active at any given time. However, this model is only given as an example. The RPC protocol makes no restrictions on the concurrency model implemented, and others are possible. For example, an implementation may choose to have RPC calls be asynchronous, so that the client may do useful work while waiting for the reply from the server. Another possibility is to have the server create a separate task to process an incoming call, so that the original server can be free to receive other requests.

There are a few important ways in which remote procedure calls differ from local procedure calls:

1. Error handling: failures of the remote server or network must be handled when using remote procedure calls.

2. Global variables and side-effects: since the server does not have access to the client's address space, hidden arguments cannot be passed as global variables or returned as side effects.

3. Performance: remote procedures usually operate one or more orders of magnitude slower than local procedure calls.

4. Authentication: since remote procedure calls can be transported over unsecured networks, authentication may be necessary. Authentication prevents one entity from masquerading as some other entity.

The conclusion is that even though there are tools to automatically generate client and server libraries for a given service, protocols must still be designed carefully.

3.1.1.1 Transports and Semantics

The RPC protocol can be implemented on several different transport protocols. The RPC protocol does not care how a message is passed from one process to another, but only with specification and interpretation of messages. However, the application may wish to obtain information about (and perhaps control over) the transport layer through an interface not specified in this document. For example, the transport protocol may impose a restriction on the maximum size of RPC messages, or it may be stream-oriented like TCP with no size limit. The client and server must agree on their transport protocol choices.

It is important to point out that RPC does not try to implement any kind of reliability and that the application may need to be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP, then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP, it must implement its own time-out, retransmission, and duplicate detection policies as the RPC protocol does not provide these services.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution requirements. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP. If an application retransmits RPC call messages after time-outs, and does not receive a reply, it cannot infer anything about the number of times the procedure was executed. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regnant them in order to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC message. The main use of this transaction ID is by the client RPC entity

in matching replies to calls. However, a client application may choose to reuse its previous transaction ID when retransmitting a call. The server may choose to remember this ID after executing a call and not execute calls with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a "reliable" transport such as TCP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume that the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP is perhaps a natural transport for RPC. RPC uses both TCP and UDP transport protocols [24].

3.1.1.2 Binding and Rendezvous Independence

The act of binding a particular client to a particular service and transport parameters is not part of this RPC protocol specification. This important and necessary function is left up to some higher-level software.

Implementers could think of the RPC protocol as the jump-subroutine instruction ("JSR") of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the binding software makes RPC useful, possibly using RPC to accomplish this task.

3.1.1.3 Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service, and vice-versa, in each call and reply message. Security and access control mechanisms can be built on top of this message authentication. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used.

3.1.1.4 RPC Protocol Requirements

The RPC protocol must provide for the following:

1. Unique specification of a procedure to be called.

2. Provisions for matching response messages to request messages.

3. Provisions for authenticating the caller to service and vice-versa.

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

1. RPC protocol mismatches.

2. Remote program protocol version mismatches.

3. Protocol errors (such as misspecification of a procedure's parameters).

4. Reasons why remote authentication failed.

5. Any other reasons why the desired procedure was not called.

3.2 An Overview of Common Object Request Broker Architecture (CORBA)

3.2.1 Overview of CORBA

CORBA (Common Object Request Broker Architecture) is a distributed object framework proposed by a consortium of over 800 companies called the Object Management Group (OMG), founded in 1989. The member organizations are ranging from larger companies to smaller ones. Some of the members are Sun Microsystems and Inprise Software Corporation [27].

CORBA serves as a specification of middleware for distributed objects. The specification does not state how the implementation should be done; there are several commercial products that implement the CORBA standard. This has both advantages and disadvantages. Many companies cooperate and share their experience which contributes to a better and improved standard. It also has its drawbacks when multiple vendors' implementations are about to communicate, which is very common in a distributed environment. Vague specifications force vendors to draw their own conclusions which give many different implementations. As the specifications become clearer, the different implementations converge and become more compatible. A CORBA-based program from any vendor, on almost

any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

CORBA uses object orientation in its architecture. The objects are pieces of running software that can live anywhere on a network. Its platform, location and implementation are of no interest for the client; in fact those details are hidden for the user. What the client is interested in is the interface of its server object. This interface is the handshake between clients and servers. CORBA uses IDL (Interface Definition Language) to define these contracts as described in. The CORBA IDL is a declarative language, which means that it contains no implementation details. It is independent of the programming language and maps to programming languages via a set of OMG standards. OMG has developed standardized mappings for C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript. When compiling the IDL interface it generates the client stub and the server skeleton. On top of the IDL stub the programmer implements the client and on the IDL skeleton the object implementation. The stubs and skeletons serve as proxies for clients and servers, respectively.

Figure 2.2 below shows a very simple view of a client making a request. A client that wishes to perform an operation on the object sends a request. The client's request is passed through the stub on the client side and continues via an ORB (Object Request Broker) and the skeleton on the implementation side to the object where it is executed. The ORB is the core or the "heartbeat" of the CORBA system. Its responsibility is to find an object implementation for the request, prepare the object implementation to receive the request and to pass the data that makes the request. It serves as a bus for objects and lets them make requests and receive responses from other objects located locally or remotely [6].

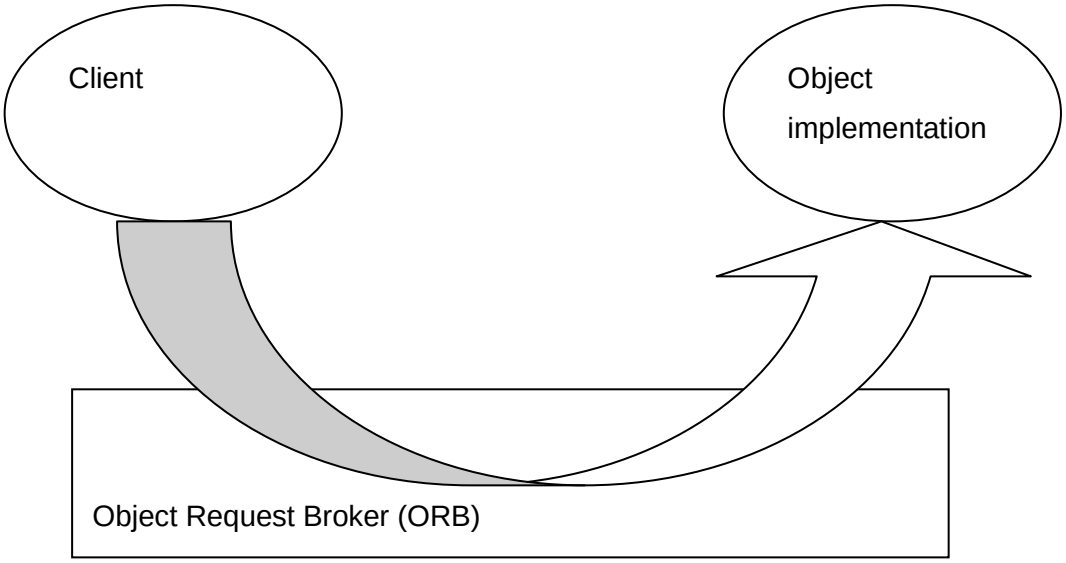

Figure 2.2 – A request is passed from a client to an object implementation

The details of the functionality and architecture behind this request have been studied in three different layers:

The top layer: The programmers view and the basic programming architecture.

The middle layer: The remote architecture, that provides the client programmer with the freedom not to have to know where an object is located in a network.

The bottom layer: The wire protocol architecture.

As mentioned above, there are several ORB implementations among which the Inprise Visibroker ORB is one of the most widely used ORBS in internet technology. This ORB is the one that has been studied in more detail.

3.2.2 The Layer Structuring in CORBA

Based on the RPC structure described in Section 2.1, three layers are used to describe the architecture of CORBA. The first layer, the top layer, referred to as the basic programming architecture, describes the programmers' view of CORBA. The middle layer, the remote architecture, describes the required infrastructure to give both the client and the server the illusion that they reside in the same address space. The last layer, the bottom layer, describes the wire protocol architecture necessary for supporting the client and the server to run on different machines. Looking at method invocations and object activation, the different parts of the CORBA architecture are described at the three layers [6].

The layer structuring in CORBA is described using the ORB implementation from Inprise called Visibroker, based on CORBA specification 2.3. Describing an actual implementation of the specification for CORBA makes the description more concrete and hopefully easier to understand.

3.2.2.1 The Top Layer

This layer corresponds to the programmers' view of the CORBA architecture. The figure 2.3 below shows the top layer of the CORBA architecture.

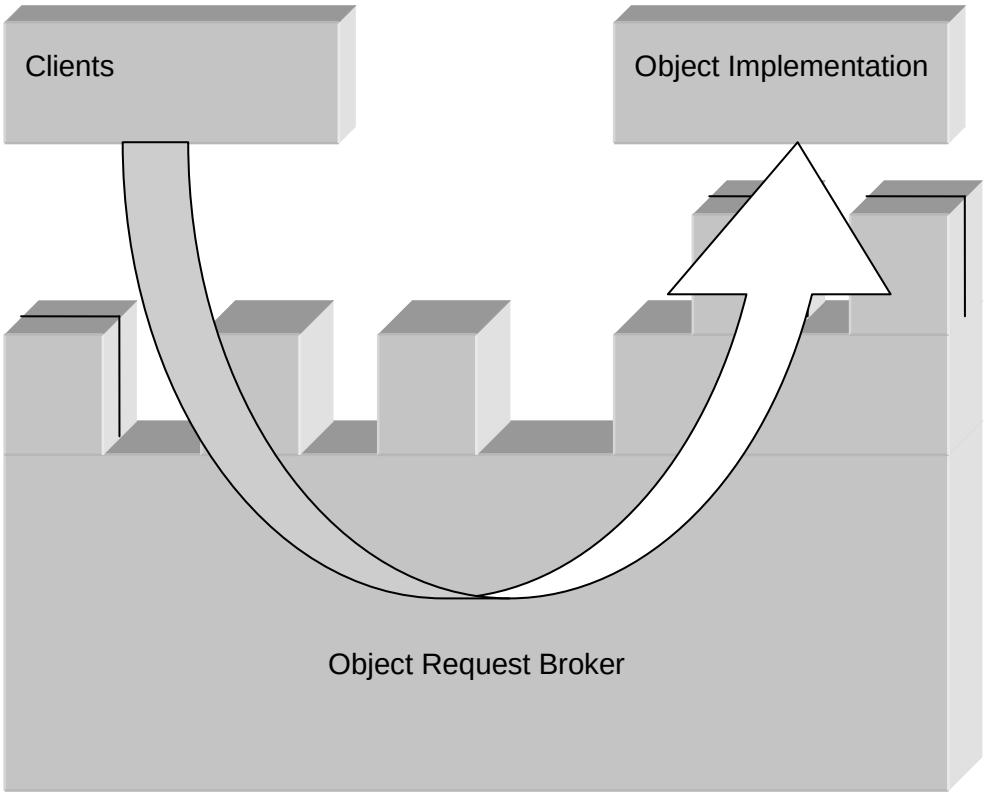

Figure 2.3 - A request is passed from client to object implementation at the top layer

Client side

On the client side, the client requests a CORBA object and invokes its methods. For a client to invoke methods on an object, it first needs a handle to the object. In CORBA, this handle is called an object reference. The process to obtain an object reference is called binding. The details of the connection between the client and the

server are totally hidden from the programmers and the client and server interact as if they reside on the same machine. The client only sees the object's interface, no implementation details. Clients interact with a CORBA object by invoking its methods described in the Interface Definition Language (IDL). The IDL serves as contract between the client and the server.

The client's request does not pass directly from the client to object implementation; it is passed via the Object Request Broker (ORB) to the server. The ORB works as a communication link between the client and server.

Server side

The server creates an instance of an object and makes it available to the client by registering it with the ORB.

3.2.2.2 CORBA Object

As stated in [6], a CORBA object is a blob of intelligence that can live anywhere on the network. It is packaged as a binary component, which clients can invoke via method invocations. There are four keys which help to describe a CORBA object:

- Encapsulation. An encapsulated CORBA object consists of two parts: the interface and its implementation. The interface is presented for the clients and the implementation is kept private. The interface serves as a contract between the client and server.
- Inheritance. Inheritance saves a lot of work for the programmers. CORBA uses inheritance in the IDL, interface inheritance.
- Polymorphism. Polymorphism is to have some operations that belong to more than one kind of object. A client invoking the same operation on a set of objects results in different things happening.
- Instantiation. Instantiation of a CORBA object is the creation of a new individual object instance.

By publishing its interface to the outside world, the CORBA object shows its methods and makes it available for requests. Clients do not need to know where the object resides and what operating system it executes on. Also the details of how the object is implemented are of no interest for the client.

The client acts as if the object always exist, maintaining its state between invocations. In reality computing resources may not be allocated for an object until

an invocation comes in. The allocated resources may be deallocated after the invocation has completed. The state between invocations is maintained on persistent storage and is loaded on activation.

3.2.2.3 Object Reference

An object reference is the reference clients use to connect to an object. Every CORBA object in a system has, regardless of its lifetime, its own object reference. The object reference is valid until the object is explicitly deleted. The lifetime of a CORBA object depends on its purpose. A CORBA object representing a main account object for an enterprise's net worth must outlive the enterprise's existence, while a CORBA object representing a shopping cart on web shop only outlives a shopping round. In a Portable Object Adaptor (POA) based ORB, the object reference is created at the server side, since its server related. The ORB described later in this section assigns the reference at object creation and the persistent services use it to save an object's state so that it can be reactivated at a later time.

The name "object reference" is very generic, when different ORBs are about to communicate the "Interoperable Object Reference" (IOR) is a better name. The IOR is understood by different ORBs, which interoperate using the protocol Internet Inter-ORB Protocol (IIOP), described at the middle layer.

The contents of the object reference are of no interest for the client, in fact only the ORB can change it. To really explain the object reference, details hidden from the programmer are required. The object reference must contain enough information for the client-side ORB to find back to server code. Without diving into details at this layer, according to [7], an object reference contains three pieces of information: An address and two pieces that are important for the server programmer:

1. An address. The address is required so that the client ORB can find the right machine.

2. The name of the POA. The name uniquely identifies a POA. A request from a client will come to the same POA that created the object reference. The POA serves as an intermediary between the implementation of an object and the ORB, for more details read about the POA at the middle layer.

3. The object Id. This used by the POA to identify the object implementation corresponding to the request.

A client can stringify an object reference using the method object_to_string(). This stringified object reference can be stored and reactivated at a later time using the method string_to_object(). The stringified version is valid as long as the object it refers to has not been deleted. All ORBs must provide the same language mapping to an object reference for a particular programming language. This makes it possible to pass a stringified object reference to any other instance of the same vendor's ORB and in addition to any other IIOP ORB. Since these stringified references can be passed by email, storage in databases and even by fax, it is the most popular way of passing object references. In Visibroker, an IOR can also be associated with an URL in the form of a string in a file. This feature is called the URL Naming Service, which allows clients to locate objects using an URL.

3.2.2.4 Interface Definition Language (IDL)

The interfaces can be defined statically using an interface definition language, called the OMG Interface Definition Language (OMG IDL). The IDL serves as a contract between clients and the associated object services. It defines the objects methods and the parameters to those operations. An interface in IDL is equivalent of a class in C++ or an interface in Java and it obeys the same lexical rules as in C++. IDL interfaces can be written in and invoked from any language, supporting CORBA bindings. This means that client and server objects written in different languages can interoperate. Some of the mappings to programming languages are shown in figure 2.4 [6].

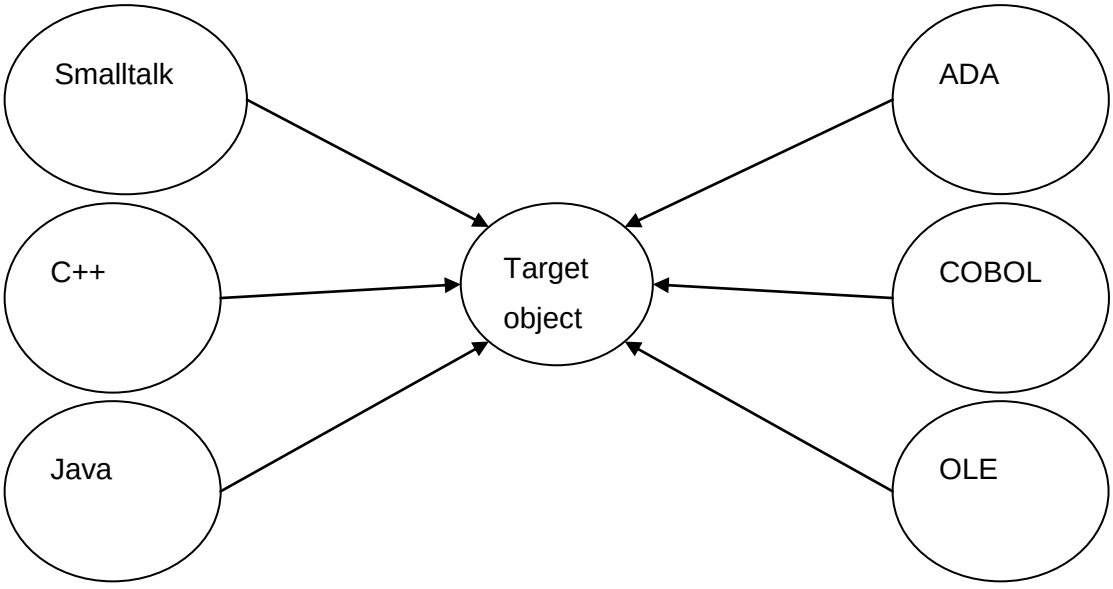

Figure 2.4 - IDL language mappings

The IDL is a purely declarative language, which means that it does not provide implementation details; it separates the interface and the implementation. One of the most important features of IDL is that it supports single and multiple inheritances. Using this, an interface can be derived from one or more existing interfaces, saving a lot of work for the programmer. OMG IDL can also specify exceptions. After the IDL is defined, vendor-specific tools can be utilized to generate the client side stubs and the server-side skeletons, which are used when a request is passed from client to server. More details are available at the middle layer.

3.2.2.5 Object Request Broker (ORB)

The ORB is used when a request is sent by a client that wishes to perform an operation on an object. The ORBs responsibility is to find an object implementation for that request, prepare the object implementation to receive the request and to pass the data that makes the request. The ORB serves as a communication link between the client and the server. The interface the client is presented is independent of where the object is located and what programming language it is implemented in.

An ORB provides a variety of distributed middleware services. It lets objects discover each other at run time and invoke each other's services. The best way to describe the ORB is to describe some of the middleware features it provides:

- Static and dynamic method invocations. The ORB lets the programmer define the methods at compile time or dynamically at run time.
- High-level language bindings. The ORB makes it possible to use different languages to implement server objects. It is possible to call objects across language and operating system boundaries.
- Self-describing system. CORBA provides run-time metadata for describing every server interface that the system has knowledge about. This helps clients to invoke services at run-time and helps tools generate code "on-the-fly".
- Local/remote transparency. An ORB has the capability to run standalone on a laptop or it can be interconnected with other ORBs using the protocol IIOP. An ORB can manage interobject calls within a single process, multiple processes within the same machine or across networks and operating systems. Either way it is totally transparent to the objects.

- Built-in security and transactions. The messages produced by the ORB include context information to handle security and transactions across machine and ORB boundaries.
- Coexistence with existing systems. The separation of interface and implementation is useful when integrating existing applications. Even if the application is implemented in stored procedures, the programmer can make it look like an object on the ORB.

Object Activation at the Top Layer Using Visibroker

Client side

1. The client explicitly initializes the ORB.
2. To obtain a reference to the remote object, the client calls the static bind () method.

Server side

1. The server explicitly initializes the ORB.
2. The POA is created and configured.
3. The POA manager is activated to tell the ORB that it is ready to accept client requests.
4. Objects are activated.
5. The server waits for client requests.

Initialize the ORB

The ORB provides a communication link between client requests and object implementations. Both sides must initialize it before communicating with it.

Create and Set-up the POA

The POA decides which servant that should be used for a client request. The following steps describe the way setting up the POA with a servant:

- Obtain a reference to the rootPOA. The rootPOA is the default POA that always is created. The rootPOA's policies are predefined and cannot be changed. A policy is an object that controls the behavior and the objects the

POA manages. To create a new POA with other policies than the default rootPOA, a server application must get a reference to the rootPOA.

- Define the POA policies for the new POA. An example of a policy is the lifespan policy, which specifies how the POA should control the lifecycle of an object implementation. The lifespan policy can be set to transient or persistent. The transient policy means that an object cannot outlive the POA that created it. The persistent policy means that the object can outlive the process in which it was created. A request invoked on a persistent object may result in a reactivation of the whole environment required for the object.
- Create a new POA with the defined policies as a child of the rootPOA.

Activate the POA through its Manager

The POA manager is default in the holding state, i.e. all requests coming from the client are queued. To activate the POA, the POA manager's state is changed from a holding state to an active state.

Activate Objects

Objects can be activated in several ways:

- Explicit: All objects are activated upon start-up.
- On-demand: The servant manager, described at the middle layer, activates an object upon receiving a request.
- Implicit: Objects are implicitly activated by the server in response to an operation by the POA.
- Default servant: The POA uses the default servant to serve client requests. This means that the same servant is used for all requests.

3.2.2.6 Wait for Requests

Once the POA is set up, the server can wait for client requests. This will run until the server is terminated.

3.2.2.7 Method invocation at the top layer using Visibroker

Client side

The client invokes a method on the remote object using the retrieved object reference.

Server side

The request is processed by the server.

3.2.2.8 The Middle Layer

The figure 2.5 below shows the architecture at the middle layer, necessary for providing the client and the server with the illusion that they are in the same address space [6]. The arrows in the figure 2.5 represent the interfaces between ORB components and its clients and object implementations.

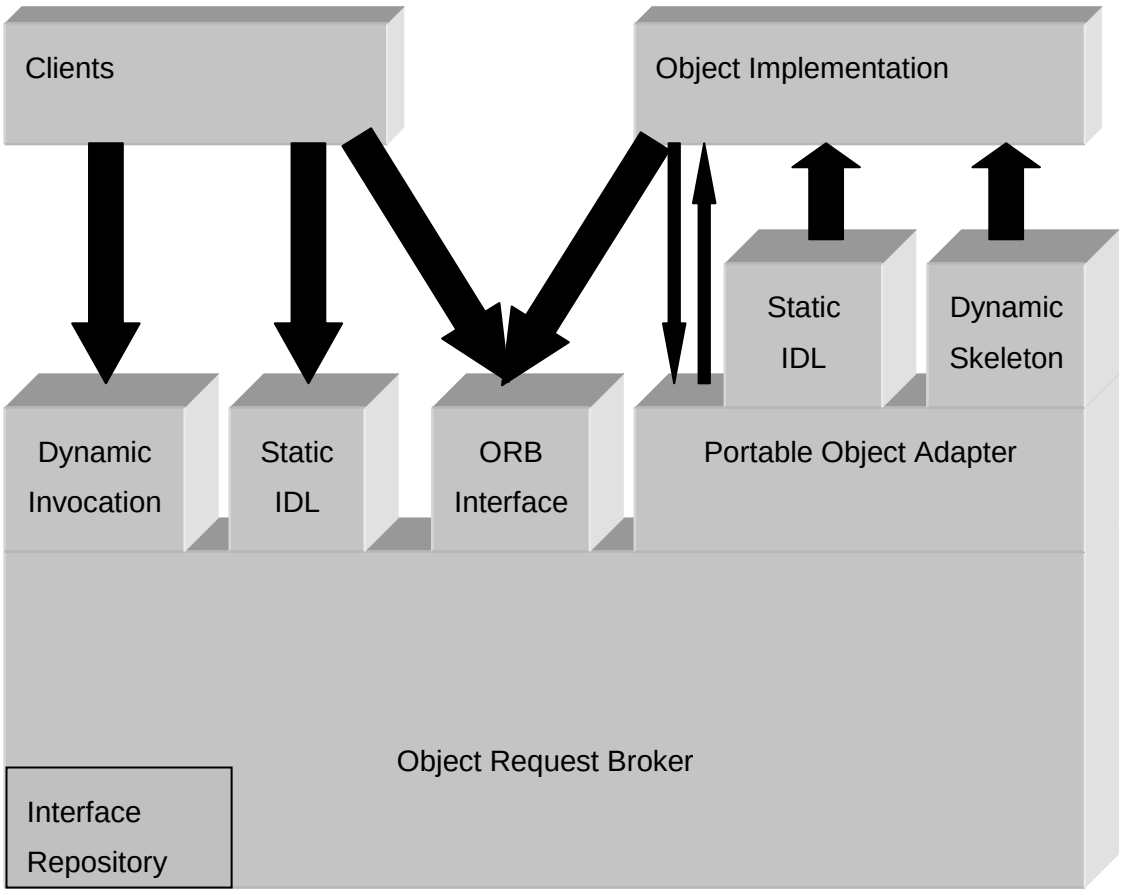

Figure 2.5 - Structure of the Object Request Broker (ORB) with clients and object implementations at the middle layer

Client side

When a client initiates a request, it retrieves the objects interface from the interface repository, which provides a secure, stateful, persistent memory for interface definitions. Once the client has found the object's interface it searches the implementation repository for the object's implementation. The interface and implementation repositories can be accessed directly via the ORB interface and indirectly through method invocations via the Static Invocation Interface (SII) and Dynamic Invocation Interface (DII). The Static Invocation Interface is the static client

IDL stubs, which are generated at compile time. The Dynamic Invocation Interface discovers methods that can be invoked at runtime.

Server side

Upon receiving the request from the client, the ORB calls the server using the static server IDL skeletons or the Dynamic Skeleton Interface (DSI). The Dynamic Skeleton Interface can deliver requests to object implementations, which have not been connected via static stubs at compile time.

The ORB acts as an object bus, as a link between a client and a server. On top of the ORB is the Object adapter (OA), which provides the run-time environment for instantiating server objects, passing the requests and assigning them object IDs. One OA that the ORB uses as the basic-handle to communicate with object services is the Basic Object Adapter (BOA). This BOA has recently been replaced by the Portable Object Adapter (POA) to provide portability on the server side, which means that one server implementation written for one ORB is portable to other ORB products.

3.2.2.9 Interface Repository (IR)

The interface repository (IR) allows obtaining and modifying the description of all the registered objects interfaces, the methods they support, and the parameters they require. It manages and provides access to a collection of objects specified in IDL. According to the CORBA specification, an ORB can use the object definitions provided by the IR to:

- Provide type-checking of request signatures, whether the request was issued through the Dynamic Invocation Interface or through a stub.
- Provide assist in checking the correctness of interface inheritance graphs.
- Provide interoperability between different ORB implementations.

For example, the information maintained in an IR is also helpful for clients and objects to:

- Manage the installation and distribution of interface definitions.
- Browse or modify IDL during development process.

Identification of an IR

The CORBA specification specifies that an ORB at least can access one IR, so multiple ORBs may share a particular IR and an ORB may access multiple IRs. This is possible because every IR has it own unique RepositoryID, which helps the ORB to keep track of it.

Usage of an IR

The CORBA specification allows the IR to be implemented by the ORB vendor, so that it suits their platform and operating system. Therefore the utilities provided by the ORB vendor are mostly used for accessing the IR.

Implementation Repository

Once the ORB has found the objects interface it searches the implementation repository for that objects implementation. The implementation repository allows the ORB to locate and activate implementations of objects. It provides a run-time repository of information about the classes a server supports, objects instantiated and their IDs. It also serves as a place to store additional information associated with the implementation of the ORB, for example debugging information, administrative control, resource allocation etc. In Visibroker the implementation of the implementation repository is called the Object Activation Daemon (OAD).

The OAD provides another service besides those provided by a typical implementation repository; if an object implementation is registered with the OAD it is automatically activated when a client attempts to access it. Activation information about all object implementations registered with the OAD is stored in the implementation repository. The OAD is an optional feature. It is a separate process that only needs to be started on those hosts where object servers are to be activated on demand. If no OAD is used the Smart Agent handles all location of objects. The Smart Agent is a dynamic, distributed directory service. If the OAD is used, it cooperates with the Smart Agent to make a connection to objects. Object implementations are registered with the OAD so that they can be activated automatically. Such objects are registered with the Smart Agent in a fashion that makes the Smart Agent believe that the objects are active and located within the OAD. When the Smart Agent receives a client request to such an object the request is forwarded to the OAD, which then directs the request to the real spawned server.

3.2.2.10 Static Invocation Interface (SII)

The Static Invocation Interface is the static client IDL stub, which is generated at IDL compile time by vendor specific tools. SII requires that the object type and the operation are defined statically, i.e. at compile time.

For the client, the stub is a proxy for a remote server object and the client must have an IDL stub for each interface it wants to use. The stub is responsible for the marshalling of the operation and its parameters when passing requests to the server.

3.2.2.11 Dynamic Invocation Interface (DII)

The DII gives the client the opportunity to invoke any operation on any object that it may access across the network. Objects for which the client has no stub or objects newly added or discovered are available for the client through DII. The DII allows synchronous, asynchronous and deferred synchronous invocation semantics. They are described here:

- Synchronous semantics: Synchronous calls block until the ORB can deliver to the client a response and result from the method invocation or an exception.
- Asynchronous: A call does not block. A response is not given to the client from the object implementation.
- Deferred synchronous: A nonblocking call with a returned result.

The object implementation cannot distinguish between an invocation that came via the SII and an invocation that came via the DII, because the ORB prepares a dynamic request so that it looks like a static request. The client chooses SII or DII, the ORB prepares the request and the object implementation does not see the difference. The dynamic binding is a great feature but there are disadvantages with it:

- The programming becomes more difficult.
- Invocations take longer time because more work is done at runtime.

Steps for dynamic invocation:

1. Identify and obtain a reference to the target object. When using the DII the traditional bind () operation is not used, because the class definition may not have been known to the client at compile time.

2. Create a Request object for the target object. A request object is created to represent each method invocation on one CORBA object. It is created transparently when using invocation via the static client stub. This object now has to be created by client programmers.

This can be done in two ways:

- Invoke the target object's _request method.
- Invoke the target object's _create_request method. This way is more complicated but has a better performance.

3. Initialize the request parameters and the result to be returned. When using the request method, each argument is added using the request's add_value method for a method that is to be invoked. The return type is set calling the set_return_type method. When using the _create_request method, the arguments, return types and exceptions are all specified when calling the _ create_request method.

4. Invoke the Request and wait for the results.

- The easiest way to invoke a request is to call its invoked method.
- The send_deferred method may be used to send a non-blocking request.
- The send_oneway method can be used to send an asynchronous request.
- A sequence of requests can be sent using the methods send_multiple_requests_oneway or send_multiple_request_deferred. The sequence of DII requests is created using an array of request objects.

5. Retrieve the results.

When using the send_deferred method, the following methods are used:

- The method poll_response. It is used to determine when the response is ready.
- The method get_response. It blocks until a response is received. When multiple requests have been carried out, the methods poll_next_response and get_next_response are used to retrieve the results.

3.2.2.12 IDL Skeletons

The static IDL skeletons are generated at compile time by vendor specific tools. They provide static interfaces for the services exported by the server. Like the IDL stubs on the client side, the skeletons do the marshalling and unmarshalling of a

request when transferring it to and receiving the result from the object implementation.

Dynamic Skeleton Interface (DSI)

The Dynamic Skeleton Interface (DSI) allows dynamic handling of object invocations so that object servers can create object implementations at run time to service client requests. The DSI is the counterpart to Dynamic Invocation Interface on the server side. Figure 2.6 below shows a request being delivered through the DSI.

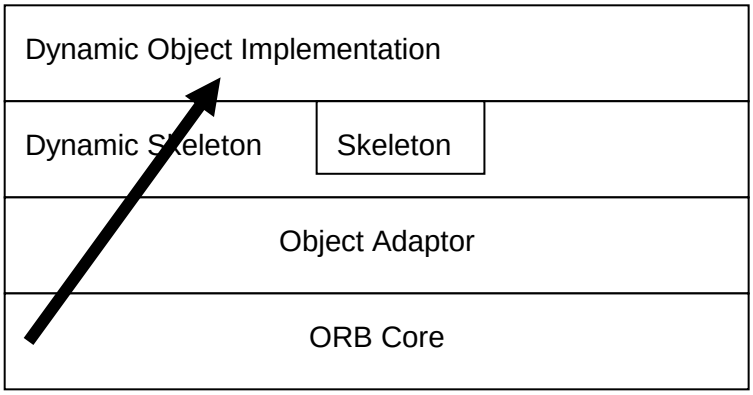

Figure 2.6 - Request delivered through dynamic skeleton

Normally an implementation is derived from a skeleton class, generated by a compiler. The DSI is a way to deliver requests from an ORB to an object implementation that does not have compile-time knowledge of the type of object it is implementing. The DSI lets an object register itself with the ORB and lets it receive and process requests from a client, without inheriting from a skeleton class.

To a client, an object implementation using DSI looks the same as any other ORB object and the client programmer does not need to provide any special code. In fact, a client cannot tell if the implementation is using the DSI or a type-specific skeleton. On the server side, using the DSI involves more manual programming when implementing the server objects than when inheriting from a skeleton class.

DSI in Visibroker

At a client request, the ORB calls the objects invoke method and sends a ServerRequest object as in parameter. The ServerRequest object represents the operation request and informs the object implementation among other things about the name of the requested method, the parameter list and how to return a result.

The object implementation is responsible for interpreting this information, call the method and fulfill the request. The object implementation is derived from the DynamicImplementation class instead of the skeleton class.

3.2.2.13 Object Adapter (OA)

An object adapter is a mechanism that connects a request using an object reference with the right code to service the request. An object implementation primarily uses the object adaptor to access services provided by the ORB. Some of these services are:

- Method invocation
- Security of transactions
- Object and implementation activation and deactivation
- Generation and interpretation of object references
- Registering implementations

Instead of using a single interface for all object implementations, the ORB can use object adaptors to target different groups of implementations. These group-targeted interfaces are more reliable and efficient than one single interface.

Basic Object Adaptor (BOA)

The Basic Object Adaptor (BOA) has recently been replaced by the Portable Object Adaptor (POA). The specification for the BOA was from the beginning vague, which led to different implementations from different vendors. These implementations were not portable on the server-side, i.e. one server implementation written for one ORB was not portable to other ORB products. In Visibroker 4.1, the BOA has been replaced by the POA which provides portability of server code. The BOA is still supported.

Portable Object Adaptor (POA)

The POA introduces portability on the server side replacing the BOA. It serves as”glue” between object implementations and the ORB. The POA is an object, which is created, has an object reference, is invoked and destroyed like other objects. What differs it from other objects is that it is locality constrained. This means that its reference cannot be passed to other computers because its job is to deal with requests on a particular computer. Besides from connecting the client’s request to a

servant, the POA is also a part of the object implementation. The implementation of an object is the combination of a POA and a servant.

The POA delivers the requests to the servant. Servers can support multiple POAs, using multiple child POAs. One POA is always present, the rootPOA, which is created automatically. All child POAs derive from the rootPOA. The figure 2.7 below shows an overview of the POA architecture used in Visibroker [6].

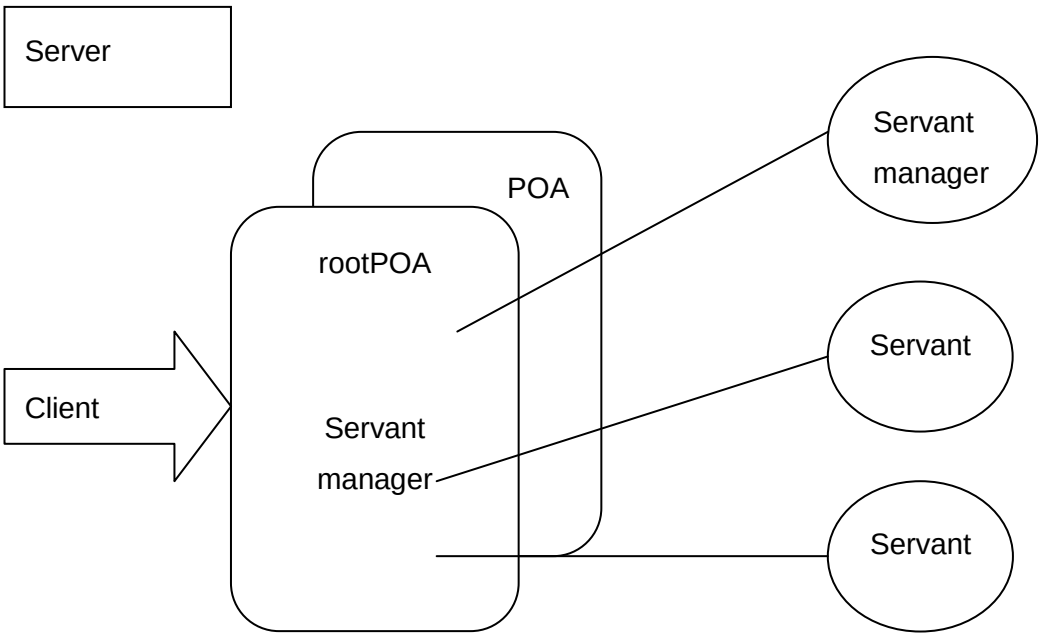

Figure 2.7 - An overview of the POA architecture used in Visibroker

Servant managers identify and assign servants to objects for the POA. Once the object has been assigned a servant, it is called an active object. The servant now associates the active object with an abstract CORBA object. Every POA is equipped with an Active Object Map, which keeps track of the Object Ids of active objects and their associated active servants.

Servant

A servant is the code written by the programmer that contains the business logic, but that is not the CORBA object itself. A servant is an active instance of the class implementing the business logic. In Java, a servant is an instance of a class. The Active object Map is used when a client request is received. If the object Id is not in the map, then it is the servant manager's job to find and activate the servant.

Servant manager

This object is responsible for managing associations between objects and servants. It also determines whether an object exists. A servant manager is optional; all objects may be loaded at start-up. A servant manager perform two types of operations, it finds and returns a servant and deactivates servants. It gives the ORB the opportunity to activate an object that is not active upon receiving a request.

There are two types of servant managers:

- ServantActivator. Activates persistent objects. Servants activated by a ServantActivator are in the Active Object Map. If the servant is not in the active object map, the server manager locates it and puts the servant Id in the active object map.
- ServantLocator. Activates transient objects. To reduce the size of the Active Object Map, servants activated by a ServantLocator are not stored in the Active Object Map. This reduces the memory consumption.

The type of servant manager is set via the policies for the ORB.

Active Object Map

A table that maps active CORBA objects to servants through the use of the object Ids.

Object Id

This id is used to identify a CORBA object within the object adaptor. The object adaptor or the application assigns it. The id is unique within the object adaptor, which created it.

rootPOA

The rootPOA is created for every ORB. Multiple child POAs can be created with the rootPOA as ancestor.

POA Manager

The POA manager is an object that controls the state of the POA. Each POA is associated with a POA manager object and it can control one or several POAs.

A POA manager can have the following four states:

- Holding. When in the holding state, the POA queues all incoming requests.
- Active. When in the active state, the POA process requests.
- Discarding. When in the discarding state, the POA discards all requests that not yet have been started.
- Inactive. When in the inactive state, the POA rejects incoming requests.

Object activation at the middle layer using Visibroker with the Smart Agent

1. The POA activates the object according to the policies. This can be done in several ways as described at the top layer. When an object is activated an object reference is created and registered with the ORB.

2. When receiving the bind() call the client stub delegates the task to the ORB.

3. The ORB contacts the Smart Agent to locate a server that is offering the requested interface. When the object implementation is located a connection is established between the object implementation and the client. If the connection was successfully established, the ORB creates a proxy object.

4. The client stub returns to the client a reference to the proxy object.

Method invocation at the middle layer using Visibroker with the Smart Agent

1. The client calls a method on a CORBA object. The client stub (proxy) creates a request object, marshals the parameters and puts the message in the communication channel.

2. When the request arrives at the server, the POA finds the skeleton, rebuilds the request object and forwards it to the skeleton.

3. The skeleton uses the request object to unmarshalling the parameters. It then invokes the method, marshals the return value and the ORB builds a return value.

4. When the reply arrives at the client, the method call returns after reading the reply. The proxy then unmarshalls the return values, checks for exceptions and returns them to the client, which finishes the call.

3.2.2.14 The Bottom Layer

The bottom layer specifies the wire protocol used for the communication between the client and server running on different machines. To support inter-ORB communications between different ORB vendors, the General Inter-ORB Protocol (GIOP) was specified. The specification for the GIOP protocol can be implemented using any connection-oriented protocol. The Internet Inter-ORB Protocol (IIOP) is the most widely used implementation of GIOP using TCP/IP as transport protocol. The parameters and the return values from the method calls are marshaled using the Common Data Representation (CDR) format.

3.2.2.15 General Inter-ORB Protocol (GIOP) and Internet Inter-ORB Protocol (IIOP)

In the CORBA 1.0 specification there were no rules for how ORBs from different vendors should communicate. Therefore a client could not communicate with a server that was not written using the same ORB. To eliminate this drawback, the Global Inter-ORB Protocol (GIOP) was introduced. The GIOP is mainly built for ORB-to-ORB communication using any connection-oriented protocol. It specifies standard transfer syntax and a set of message formats. According to the specification, the GIOP was designed to meet the following goals:

- Widest possible availability. The IIOP is based on TCP/IP, the most widely used communications transport mechanism available, and defines only the minimum additional layers to transfer CORBA requests between ORBs.
- Simplicity. The GIOP was designed to be as simple as possible, while working with the other necessary goals.
- Scalability. It was designed to scale to the size of today's Internet and beyond.
- Low cost. Adding support for the GIOP/IIOP to an existing or new ORB should not require too much engineer investment.
- Generality. The GIOP was designed to be mapped onto any connection – oriented protocol.
- Architectural neutrality. The GIOP makes minimal assumptions about the architecture and implementation of the ORBs supporting it.

The GIOP consists of three specifications:

- The Common Data Representation (CDR). The CDR has the following features:
    1. Variable byte ordering. The sender decides the ordering and the receiver is responsible for swapping the bytes into the right order.
    2. Aligned primitive types. Primitive OMG IDL data types are aligned according to their natural boundaries as described in [7].
    3. Complete OMG IDL mapping. The CDR describes representations for all OMG IDL data types.
- GIOP message formats. Formats for exchanging messages between interoperating ORBs are specified. The GIOP specifies seven message formats for ORB-to-ORB communications. Message transfer is done using the transport protocol in the following ways:
    1. Asymmetric connection usage. To avoid race conditions, the client and server roles are assigned at connection. The client originates the connection and send requests, but may not send replies. The server accepts the connection and send replies, but may not send requests.
    2. Request multiplexing. Multiple clients attached to the same ORB may share a connection to a remote ORB.
    3. Overlapping requests. GIOP is designed to allow overlapping of asynchronous requests. It's up to the implementation to control the border of messages.
    4. Connection management. Messages for request cancellation and orderly connection shutdown are provided by the GIOP. This used for reclaiming and reusing connection resources.
- GIOP transport assumptions. The GIOP requires:
    1. A connection-oriented transport protocol.
    2. Reliable delivery.
    3. Participants must be notified of connection loss.
    4. Initiation of a connection must meet certain requirements.

The IIOP is the GIOP mapped onto the transport protocol TCP/IP. The IIOP is used automatically (other connection-oriented protocols can be used) when CORBA objects invoke objects on a remote server.

Object activation at the bottom layer using Visibroker with the Smart Agent

1. The POA activates the object according to the policies. The server generates an IOR, which contains a machine name, a TCP/IP port number and an object key. This reference is registered with the ORB.

2. Upon receiving the bind() request, the client side ORB locates the machine that supports the requested interface. After locating the machine, it sends a request via TCP/IP to the server side ORB.

3. When the client side receives the object reference, the proxy extracts the endpoint address and establishes a socket connection to the server.

Method invocation at the bottom layer using Visibroker with the Smart Agent

1. When receiving the request, the proxy marshals the parameters in the Common Data Representation (CDR).

2. The established socket connection is used to transfer the request.

3. The skeleton is located.

4. After invoking the server object, the return values a marshaled by the skeleton using the CDR format.

3.2.2.16 Thread management and objects by value

Threads in Inprise Visibroker

A thread is a sequential flow of control within a process. Threads are lightweight so there can be many of them within a process. By using multiple threads concurrency is provided, which increases the performance. In applications, several computations can be done simultaneously. Visibroker provides two threading policies: thread pooling and thread-per-session. These are described below.

Thread pooling

This is the default policy. A worker thread is allocated for each client request and is used by the client during that request. When the request has completed, the thread is returned to the pool. In this way the thread is reused and can be assigned to other clients' requests. A highly active client with many simultaneous requests is serviced

with many threads, ensuring that the requests are quickly executed. Less active clients can still have their requests serviced immediately by sharing a single thread. The thread pool is using dynamic allocating of threads, meaning that the number of concurrent requests decides the number of threads currently available in the pool. The pool grows when the number of concurrent requests increases and shrinks when the need for resources decreases. The size of the thread pool can be configured to meet special needs.

Thread-per-session

When using the thread-per-session management, a new thread is allocated each time a new client connects to the server. This thread handles all the requests from a particular client and is destroyed when the client disconnects from the server. The maximum number of threads to be allocated for client connections can be specified.

Objects passed by value

In the architecture described, all CORBA objects have object references and every client invokes the same copy of the object, using object by reference. From the client's point of view this is straightforward. If there is a drawback with this, it is that every invocation requires a network roundtrip for objects that resides remote from the client. For objects that represent a collection of data, many network roundtrips are required for a client using the collection. In this case it would be convenient to pass the object by value, which means packaging the whole object and send it over the wire. When the object arrives at the client it is recreated as a running object, which allows the client to make subsequent invocations on the local object. This reduces the network traffic.

As stated in [6], some say that objects by value breaks many of the CORBA transparencies, in particular, implementation and language transparency, because the IDL for the object is no longer the only contract between client and server. Instead, client and server must share some common understanding of what the methods do and how to implement the behavior of the methods.

3.3 An overview of Distributed Component Object Model (DCOM) and COM+

3.3.1 Overview of COM

The Microsoft's Component Object Model (COM) supports interaction between a client and a server object as specified in [7]. The client and the server either reside within the same address space (called in-process), or in different processes on the same host (called local or out-of-process). Their interaction is defined so that the connection between the components is invisible to the programmer; COM transparently catches the calls from the client and forwards them to the other component.

3.3.2 What are a COM client and a COM server?

This is an informal specification of a COM client and a COM server from the COM specification; "The client is any piece of code (not necessarily an application) that somehow obtains a pointer through which it can access the services of an object and then invokes those services when necessary. The server is some piece of code that implements the object and structures in such a way that the COM system can match that implementation to a class identifier, or CLSID."

3.3.3 Overview of DCOM

DCOM is an extension of the Component Object Model (COM). COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediary system component. The client calls methods in the component without any overhead whatsoever.

In today's operating systems, processes are shielded from each other. A client that needs to communicate with a component in another process cannot call the component directly, but has to use some form of interprocess communication provided by the operating system. COM provides this communication in a completely transparent fashion: it intercepts calls from the client and forwards them to the component in another process. Figure 2.8 illustrates how the COM/DCOM run-time libraries provide the link between client and component [28].

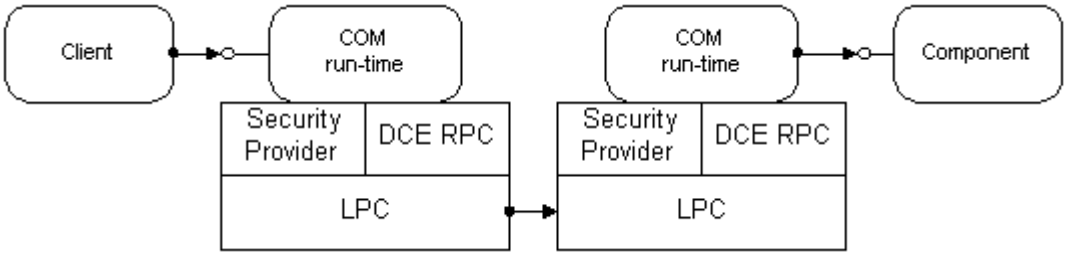

Figure 2.8 - COM components in different processes

When client and component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware that the wire that connects them has just become a little longer.

Figure 2.9 shows the overall DCOM architecture: The COM run-time provides object-oriented services to clients and components and uses RPC and the security provider to generate standard network packets that conform to the DCOM wire-protocol standard [28].

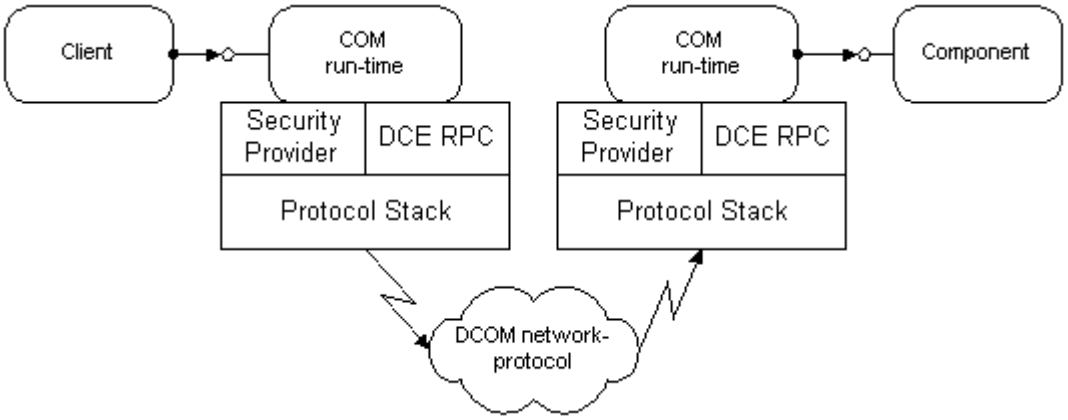

Figure 2.9 - DCOM: COM components on different machines

3.3.4 Components and Reuse

Most distributed applications are not developed from scratch and in a vacuum. Existing hardware infrastructure, existing software, and existing components, as well as existing tools, need to be integrated and leveraged to reduce development and deployment time and cost. DCOM directly and transparently takes advantage of any existing investment in COM components and tools. A huge market for off-the-shelf components makes it possible to reduce development time by integrating

standardized solutions into a custom application. Many developers are familiar with COM and can easily apply their knowledge to DCOM-based distributed applications.

Any component that is developed as part of a distributed application is a candidate for future reuse. Organizing the development process around the component paradigm lets you continuously raise the level of functionality in new applications and reduce time-to-market by building on previous work.

Designing for COM and DCOM assures that your components are useful now and in the future.

3.3.5 Location Independence

When you begin to implement a distributed application on a real network, several conflicting design constraints become apparent:

- Components that interact more should be "closer" to each other.
- Some components can only be run on specific machines or at specific locations.
- Smaller components increase flexibility of deployment, but they also increase network traffic.
- Larger components reduce network traffic, but they also reduce flexibility of deployment.

With DCOM these critical design constraints are fairly easy to work around, because the details of deployment are not specified in the source code. DCOM completely hides the location of a component, whether it is in the same process as the client or on a machine halfway around the world. In all cases, the way the client connects to a component and calls the component's methods is identical. Not only does DCOM require no changes to the source code, it does not even require that the program be recompiled. A simple reconfiguration changes the way components connect to each other.

DCOM's location independence greatly simplifies the task of distributing application components for optimum overall performance. Suppose, for example, that certain components must be placed on a specific machine or at a specific location. If the application has numerous small components, you can reduce network loading by deploying them on the same LAN segment, on the same machine, or even in the same process. If the application is composed of a smaller number of large

components, network loading is less of a problem, so you can put them on the fastest machines available, wherever those machines are.

Figure 2.10 shows how the same "validation component" can be deployed on the client machine, when network-bandwidth between "client" machine and "middle-tier" machine is sufficient, and on the server machine, when the client is accessing the application through a slow network link [28].

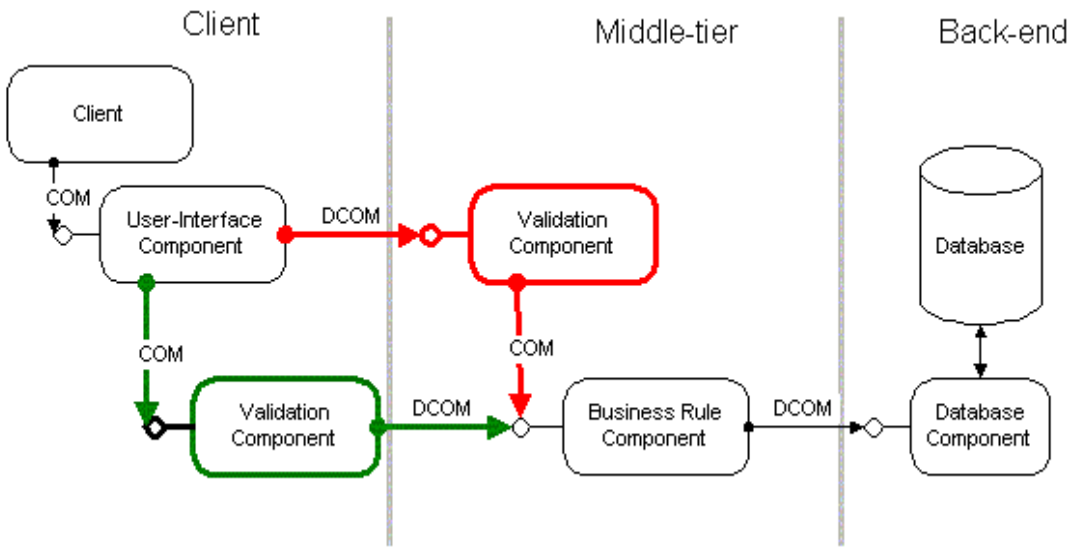

Figure 2.10 - Location independence

With DCOM's location independence, the application can combine related components into machines that are "close" to each other onto a single machine or even into the same process. Even if a larger number of small components implement the functionality of a bigger logical module, they can still interact efficiently among each other. Components can run on the machine where it makes most sense: user interface and validation on or close to the client, database-intensive business rules on the server close to the database.

3.3.6 Language Neutrality

A common issue during the design and implementation of a distributed application is the choice of the language or tool for a given component. Language choice is typically a trade-off between development cost, available expertise, and performance. As an extension of COM, DCOM is completely language-independent. Virtually any language can be used to create COM components, and those components can be used from even more languages and tools. Java, Microsoft

Visual C++, Microsoft Visual Basic, Delphi, PowerBuilder, and Micro Focus COBOL all interact well with DCOM.

With DCOM's language independence, application developers can choose the tools and languages that they are most familiar with. Language independence also enables rapid prototyping: components can be first developed in a higher-level language, such as Microsoft Visual Basic, and later reimplemented in a different language, such as C++ or Java, that can better takes advantage of advanced features such as DCOM's free threading, free multithreading and thread pooling.

3.3.7 Connection Management

Network connections are inherently more fragile than connections inside a machine. Components in a distributed application need to be notified if a client is not active anymore, even—or especially—in the case of a network or hardware failure.

DCOM manages connections to components that are dedicated to a single client, as well as components that are shared by multiple clients, by maintaining a reference count on each component. When a client establishes a connection to a component, DCOM increments the component's reference count. When the client releases its connection, DCOM decrements the component's reference count. If the count reaches zero, the component can free itself.

DCOM uses an efficient pinging protocol to detect if clients are still active. Client machines send a periodic message. DCOM considers a connection as broken if more than three ping periods pass without the component receiving a ping message. If the connection is broken, DCOM decrements the reference count and releases the component if the count has reached zero. From the point of view of the component, both the benign case of a client disconnecting and the fatal case of a network or client machine crash are handled by the same reference counting mechanism. Applications can use this distributed garbage collection mechanism for free.

In many cases, the flow of information between a component and its clients is not unidirectional: The component needs to initiate some operation on the client side, such as a notification that a lengthy process has finished, the update of data the user is viewing (news ticker or stock ticker), or the next message in a collaborative environment like teleconferencing or a multi-user game. Many protocols make it difficult to implement this kind of symmetric communication. With DCOM, any

component can be both a provider and a consumer of functionality. The same mechanism and features manage communication in both directions, making it easy to implement peer-to-peer communication, as well as client/server interactions.

DCOM provides a robust distributed garbage collection mechanism that is completely transparent to the application. DCOM is an inherently symmetric network protocol and programming model. Not only does it offer the traditional unidirectional client-server interaction, but it also provides rich, interactive communication between clients and servers and among peers.

3.3.8 Scalability

A critical factor for a distributed application is its ability to grow with the number of users, the amount of data, and the required functionality. The application should be small and fast when the demands are minimal, but it should be able to handle additional demands without sacrificing performance or reliability. DCOM provides a number of features that enhance your application's scalability.

3.3.9 Symmetric Multiprocessing (SMP)

DCOM takes advantage of Windows NT support for multiprocessing. For applications that use a free-threading model, DCOM manages a thread pool for incoming requests. On multiprocessor machines, this thread pool is optimized to the number of available processors: Too many threads result in too much context switching, while too few threads can leave some processors idle. DCOM shields the developer from the details of thread management and delivers the optimal performance that only costly hand coding of a thread-pool manager could provide.

DCOM applications can easily scale from small single processor machines to huge multiprocessor systems by seamlessly taking advantage of Windows NT support for symmetric multiprocessing.

3.3.10 Flexible Deployment

As the load on an application grows, not even the fastest multiprocessor machine may be able to accommodate demand, even if your budget can accommodate such a machine. DCOM's location independence makes it easy to distribute components over other computers, offering an easier and less expensive route to scalability.

Redeployment is easiest for stateless components or for those that do not share their state with other components. For components such as these, it is possible to run multiple copies on different machines. The user load can be evenly distributed among the machines, or criteria like machine capacity or even current load can be taken into consideration. With DCOM, it is easy to change the way clients connect to components and components connect to each other. The same components can be dynamically redeployed, without any rework or even recompilation. All that is necessary is to update the registry, file system, or database where the location of each component is stored.

Most real-world applications have one or more critical components that are involved in most of the operations. These can be database components or business rule components that need to be accessed serially to enforce "first come, first served" policies. These kinds of components cannot be duplicated, since their sole purpose is to provide a single synchronization point among all users of the application. To improve the overall performance of a distributed application, these "bottleneck" components have to be deployed onto a dedicated, powerful server. Again, DCOM helps by letting you isolate these critical components early in the design process, deploying multiple components on a single machine initially and moving the critical components to dedicated machines later. No redesign or even recompilation of the components is needed. This is shown in figure 2.11 [28].

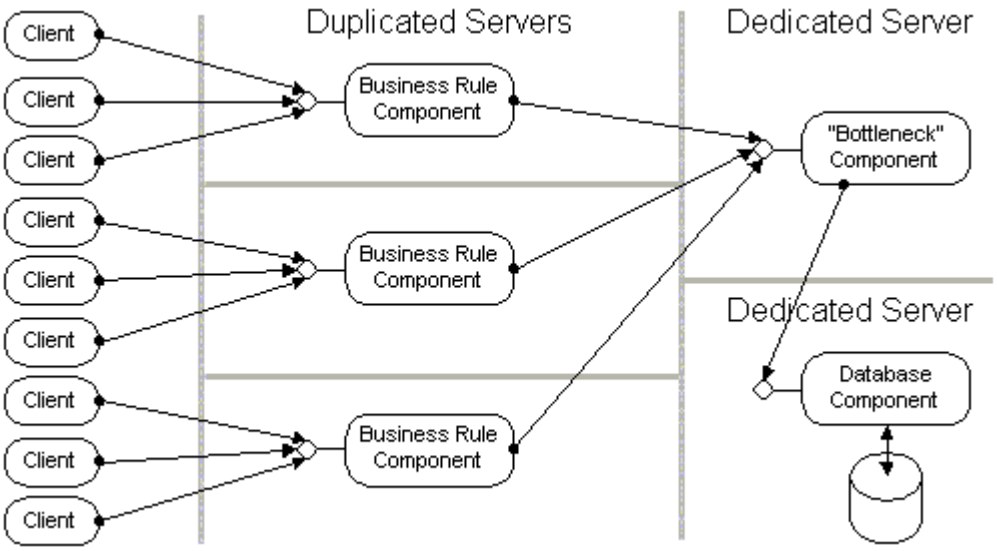

Figure 2.11 - Isolating critical components

For these critical bottleneck components, DCOM can make the overall task go more quickly. Such bottlenecks are usually part of a processing sequence, such as buy or

sell orders in an electronic trading system: Requests must be processed in the order they are received (first come, first served). One solution is to break the task into smaller components and deploy each component on a different machine. The effect is similar to pipelining as used in modern microprocessors: The first request comes in, the first component processes it (does, for example, consistency checking) and passes the request on to the next component (which might, for example, update the database). As soon as the first component passes the request on to the second component, it is ready to process the next request. In effect, there are two machines working in parallel on multiple requests, while the order in which requests are processed is guaranteed. The same approach is possible using DCOM on a single machine: multiple components can run on different threads or in different processes. This approach simplifies scalability later, when the threads can be distributed on a multiprocessor machine or the processes can be deployed on different machines.

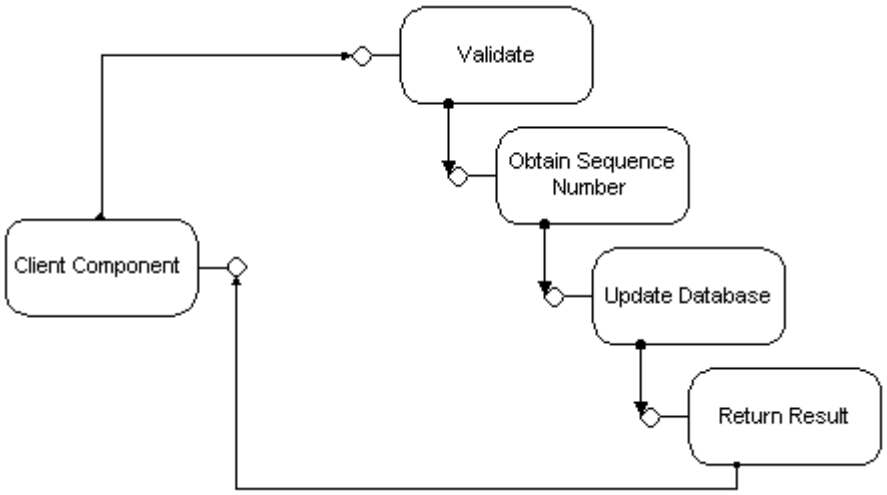

Figure 2.12 – Pipelining

DCOM's location-independent programming model makes it easy to change deployment schemes as the application grows: A single server machine can host all the components initially, connecting them as very efficient in-process servers. Effectively, the application behaves as a highly tuned monolithic application. As demand grows, other machines can be added, and the components can be distributed among those machines without any code changes like in figure 2.12 [28].

3.3.11 Evolving Functionality: Versioning

Besides scaling with the number of users or the number of transactions, applications also need to scale as new features are required. Over time, new tasks need to be

incorporated and existing ones modified. In the conventional approach, either clients or components have to be updated simultaneously or the old component has to be retained until all clients have upgraded—an undertaking that can become a major administrative burden when a significant number of geographically dispersed sites or users are involved.

DCOM provides flexible evolutionary mechanisms for clients and components. With COM and DCOM, clients can dynamically query the functionality of the component. Instead of exposing its functionality as a single, monolithic group of methods and properties, a COM component can appear differently to different clients. A client that uses a certain feature needs access only to the methods and properties it uses. Clients can also use more than one feature of a component simultaneously. If other features are added to the component, they do not affect an older client that is not aware of them.

Being able to structure components this way, enables a new kind of evolution: The initial component exposes a core set of features as COM interfaces, on which every client can count. As the component acquires new features, most (often even all) of these existing interfaces will still be necessary; and new functions and properties appear in additional interfaces without changing the original interfaces at all. Old clients still access the core set of interfaces as if nothing had changed. New clients can test for the presence of the new interfaces and use them when available, or they can degrade gracefully to the old interfaces that are shown in figure 2.13 [28].

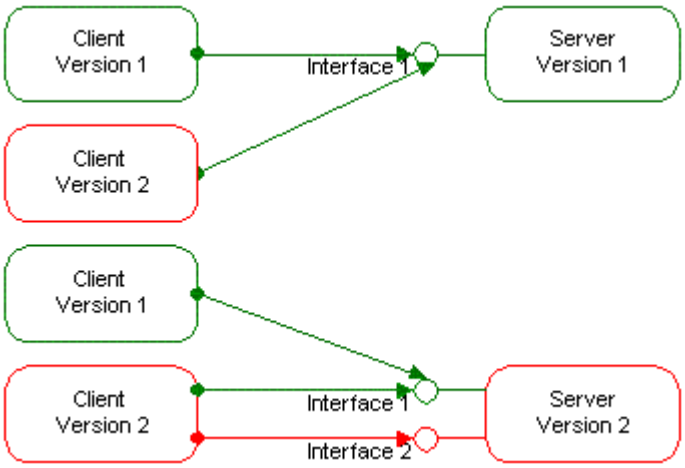

Figure 2.13 - Robust versioning

Because functionality is grouped into interfaces in the DCOM programming model, you can design new clients that run with old servers, new servers that run with old

clients, or mix and match to suit your needs and programming resources. With conventional object models, even a slight change to a method fundamentally changes the contract between the client and the component. In some models, it is possible to add new methods to the end of the list of methods, but there is no way to safely test for the new methods on old components. From the network's perspective, things become even more complicated: Encoding and wire-representation typically depend on the order of the methods and parameters. Adding or changing methods and parameters also changes the network protocol significantly. DCOM handles all these problems with a single, elegant, unified approach for both the object model and the network protocol.

3.3.12 Performance

Scalability is not much of a benefit if the initial performance is not satisfactory. It is always good to know that more and better hardware can take an application to its next evolutionary step, but what about the entry-level requirements? Don't all these high-end scalability features come at a price? Doesn't supporting every language from COBOL to assembly language necessarily compromise performance? Doesn't the ability to run a component on the other side of the world preclude running it efficiently in the same process as the client?

In COM and DCOM, the client never sees the server object itself, but the client is never separated from the server by a system component unless it's absolutely necessary. This transparency is achieved by a strikingly simple idea: the only way a client can talk to the component is through method calls. The client obtains the addresses of these methods from a simple table of method addresses (a "vtable"). When the client wants to call a method on a component, it obtains the method's address and calls it. The only overhead incurred by the COM programming model over a traditional C or assembly language function call is the simple lookup of the method's address (indirect function call vs. direct function call). If the component is an in-process component running on the same thread as the client, the method call arrives directly at the component. No COM or system code is involved; COM only defines the standard for laying out the method address table.

What happens when the client and the component are actually not as close—on another thread, in another process, or on another machine at the other side of the world? COM places its own remote procedure call (RPC) infrastructure code into the vtable and then packages each method call into a standard buffer representation,

which it sends to the component's side, unpacks it, and reissues the original method call: COM provides an object-oriented RPC mechanism.

How fast is this RPC mechanism? There are different performance metrics to consider:

- How fast is an "empty" method call?
- How fast are "real world" method calls that send and return data?
- How fast is a network round trip?

The overall performance and scalability advantages of DCOM can only be reached by implementing sophisticated thread-pool managers and pinging protocols. Most distributed applications will not want or need to incur this significant investment for obtaining minor performance gains, while sacrificing the convenience of the standardized DCOM wire-protocol and programming model.

3.3.13 Bandwidth and Latency

Distributed applications take advantage of a network to tie components together. In theory, DCOM completely hides the fact that components are running on different computers. In practice however, applications need to consider the two primary constraints of a network connection:

Bandwidth. The size of the parameters passed into a method call directly affects the time it takes to complete the call.

Latency. The physical distance and the number of network elements involved (such as routers and communication lines) delay even the smallest data packet significantly. In the case of a global network like the Internet, these delays can be on the order of seconds.

How does DCOM help applications to deal with these constraints? DCOM itself minimizes network round trips wherever possible to avoid the impact of network latency. DCOM's preferred transport protocol is the connectionless UDP subset of the TCP/IP protocol suite: The connectionless nature of this protocol allows DCOM to perform several optimizations by merging many low-level acknowledge packages with actual data and pinging messages. Even running over connection-oriented protocols, DCOM still offers significant advantages over application-specific custom protocols.

3.3.14 Shared Connection Management between Applications

Most application level protocols require some kind of lifetime management. The component needs to get notified when a client machine suffers a catastrophic hardware failure or the network connection between client and component breaks for an extended period of time.

A common approach to this problem is to send keep-alive message at periodic intervals (pinging). If the server does not receive a ping message for a specified time, it declares the client "dead."

DCOM uses a per machine keep-alive message. Even if the client machine uses 100 components on a server machine, a single ping message keeps all the clients connections alive. In addition to consolidating all the ping messages, DCOM minimizes the size of these ping messages by using delta pinging. Instead of sending 100 client identifiers, it creates meta-identifiers that represent all 100 references. If the set of references changes, only the delta between the two references sets is transmitted. Finally, DCOM piggybacks the ping message onto regular messages. Only if the entire client machine is idle with respect to a given server machine does it send periodic ping messages (at a 2-minute interval).

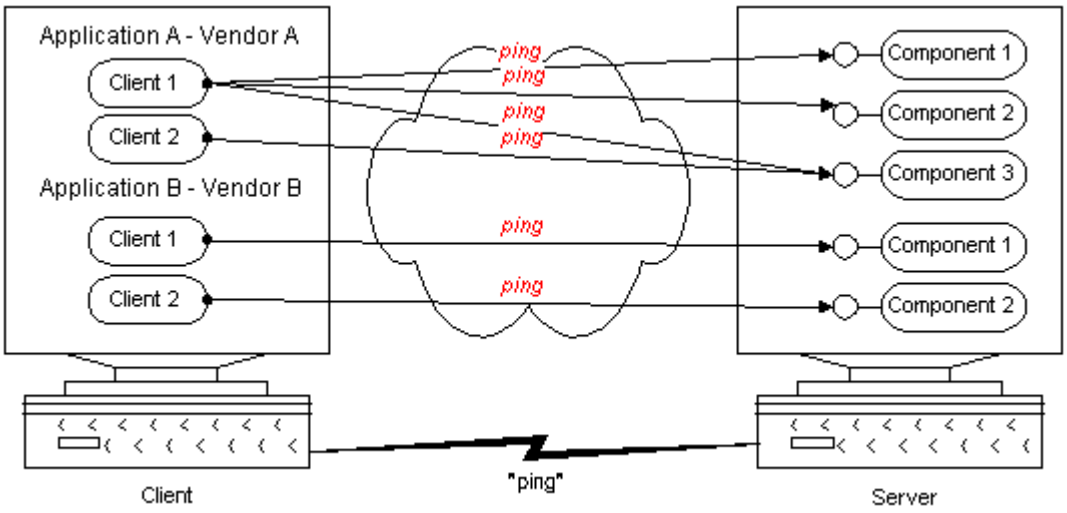

Figure 2.14 - Consolidated lifetime management

DCOM allows different applications (even from different vendors) to share a single, optimized lifetime management and network failure detection protocol, reducing bandwidth significantly that is shown in figure 2.14 [28]. If 100 different applications with 100 different custom protocols are running on a server, this server would

normally receive one ping message for each of those applications from each of the connected clients. Only if these protocols somehow coordinate their pinging strategies can the overall network overhead be reduced. DCOM automatically provides this coordination among arbitrary COM-based custom protocols.

3.3.15 Optimize Network Round-Trips

A common problem in designing distributed applications is an excessive number of network round trips between components on different machines. On the Internet, each of these round trips incurs a delay of typically 1 second, often significantly more. Even over a fast local network, round-trip times are typically measured in milliseconds—orders of magnitude above the cost of local operations.

A common technique for reducing the number of network round trips is to bundle multiple method calls into a single method invocation (batching or boxcarring). DCOM uses this technique extensively for tasks such as connecting to an object or creating a new object and querying its functionality. The disadvantage of this technique for general components is that the programming model changes significantly between the local and the remote case.

DCOM makes it easy for component designers to perform batching without requiring the clients to use a batching-style API. DCOM's marshaling mechanism lets the component inject code on the client side, called a "proxy object," that can intercept multiple method calls and bundle them into a single remote procedure call.

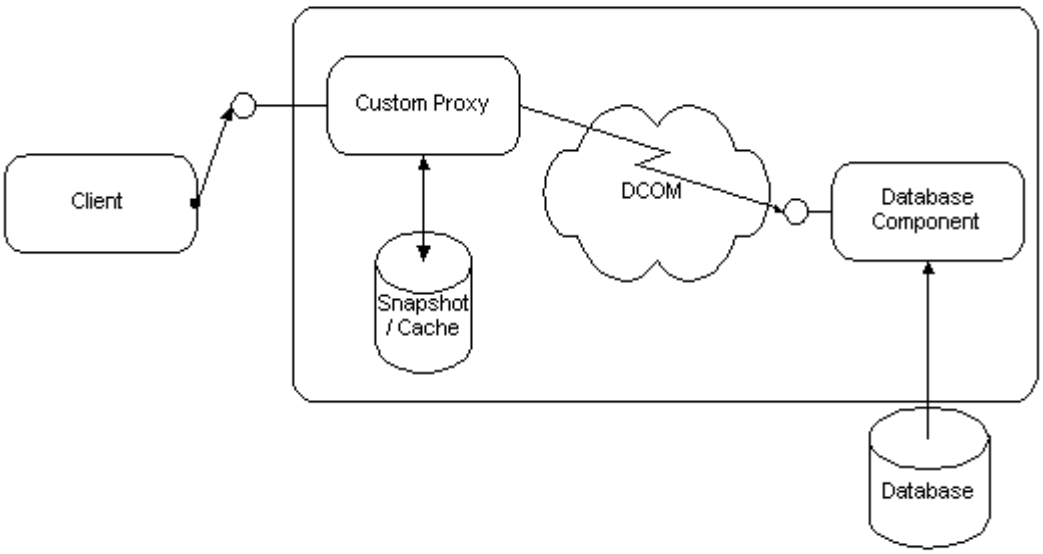

Figure 2.15 - The component model: Client-side caching

DCOM also allows efficient referrals from one component to the other. If a component holds a reference to another component on a separate machine, it can pass this reference to a client running on a third machine (refer the client to another component running on another machine). When the client uses this reference, it communicates directly with the second component. DCOM short-circuits these references and lets the original component and machine get out of the picture entirely. This enables custom directory services that can return references to a wide range of remote components. This scheme is shown in figure 2.15 and 2.16 [28].

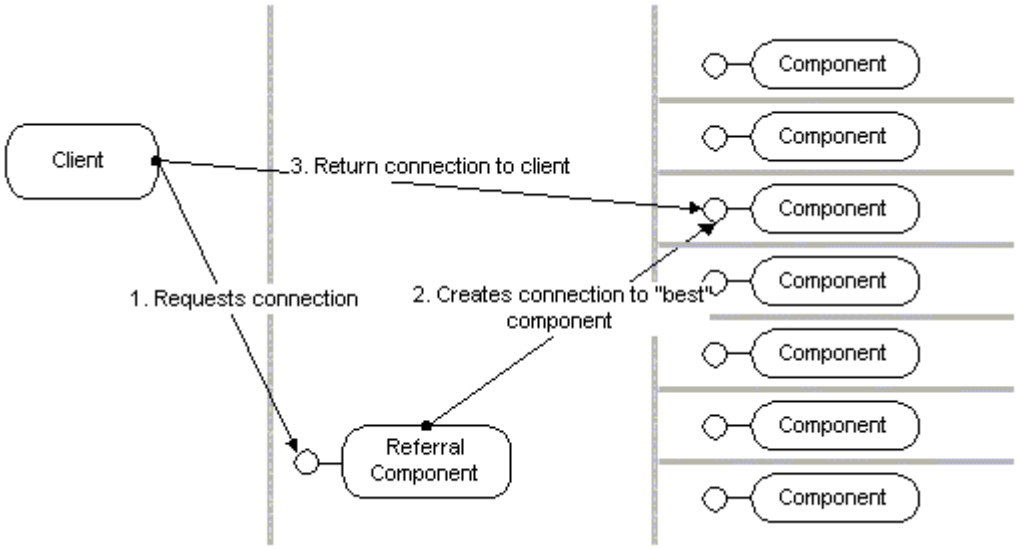

Figure 2.16 - Referral

If necessary, DCOM even allows components to plug in arbitrary custom protocols that use means outside of the DCOM mechanism which is shown in figure 2.17. The component can use custom marshaling to inject an arbitrary proxy object into the client process, which can then use any arbitrary protocol to talk back to the component.

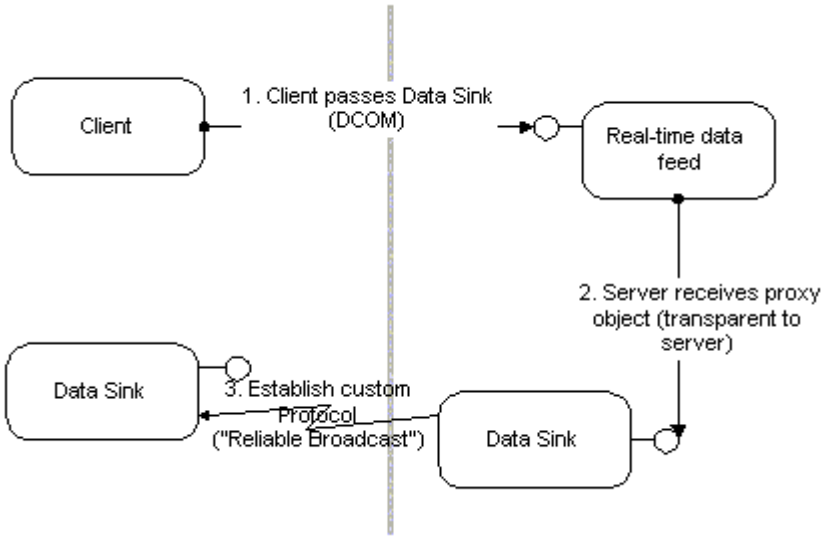

Figure 2.17 - Replacing DCOM with custom protocols

DCOM provides a multitude of ways to "tweak" the actual network protocol and network traffic without changing the way that clients perceive the component: Client-side caching, referrals, and replacing the network transport when necessary are but a few techniques that are possible.

3.3.16 Security

Using the network for distributing an application is challenging not only because of the physical limitations of bandwidth and latency. It also raises new issues related to security between and among clients and components. Since many operations are now physically accessible by anyone with access to the network, access to these operations has to be restricted at a higher level.

Without security support from the distributed development platform, each application would be forced to implement its own security mechanisms. A typical mechanism would involve passing some kind of username and password (or a public key)—usually encrypted—to some kind of logon method. The application would validate these credentials against a user database or directory and return some dynamic identifier for use in future method calls. On each subsequent call to a secure method, the clients would have to pass this security identifier. Each application would have to store and manage a list of usernames and passwords, protect the user directory against unauthorized access, and manage changes to passwords, as well as dealing with the security hazard of sending passwords over the network.

A distributed platform must thus provide a security framework to safely distinguish different clients or different groups of clients so that the system or the application has a way of knowing who is trying to perform an operation on a component. DCOM uses the extensible security framework provided by Windows NT. Windows NT provides a solid set of built-in security providers that support multiple identification and authentication mechanisms, from traditional trusted-domain security models to noncentrally managed, massively scaling public-key security mechanisms. A central part of the security framework is a user directory, which stores the necessary information to validate a user's credentials (user name, password, public key). Most DCOM implementations on non-Windows NT platforms provide a similar or identical extensibility mechanism to use whatever kind of security providers is available on that platform. Most UNIX implementations of DCOM will include a Windows NT-compatible security provider.

Before looking more closely at these Windows NT security and directory providers, let's take a look at how DCOM uses this general security framework to make building secure applications easier.

3.3.16.1 Security by Configuration

DCOM can make distributed applications secure without any security-specific coding or design in either the client or the component. Just as the DCOM programming model hides a component's location; it also hides the security requirements of a component. The same (existing or off-the-shelf) binary code that works in a single-machine environment, where security may be of no concern, can be used in a distributed environment in a secure fashion.

DCOM achieves this security transparency by letting developers and administrators configure the security settings for each component. Just as the Windows NT File System lets administrators set access control lists (ACLs) for files and directories, DCOM stores Access Control Lists for components. These lists simply indicate which users or groups of users have the right to access a component of a certain class. These lists can easily be configured using the DCOM configuration tool (DCOMCNFG) or programmatically using the Windows NT registry and Win32 security functions.

Whenever a client calls a method or creates an instance of a component, DCOM obtains the client's current username associated with the current process (actually

the current thread of execution). Windows NT guarantees that this user credential is authentic. DCOM then passes the username to the machine or process where the component is running. DCOM on the component's machine then validates the username again using whatever authentication mechanism is configured and checks the access control list for the component (actually for the first component run in the process containing the component). If the client's username is not included in this list (either directly or indirectly as a member of a group of users), DCOM simply rejects the call before the component is ever involved. This default security mechanism is completely transparent to both the client and the component and is highly optimized. It is based on the Windows NT security framework, which is probably one of the most heavily used (and optimized!) parts of the Windows NT operating system: on each and every access to a file or even to a thread-synchronization primitive like an event or semaphore, Windows NT performs an identical access check. The fact that Windows NT can still compete with and beat the performance of competing operating systems and network operating systems shows how efficient this security mechanism is in figure 2.18 [28].

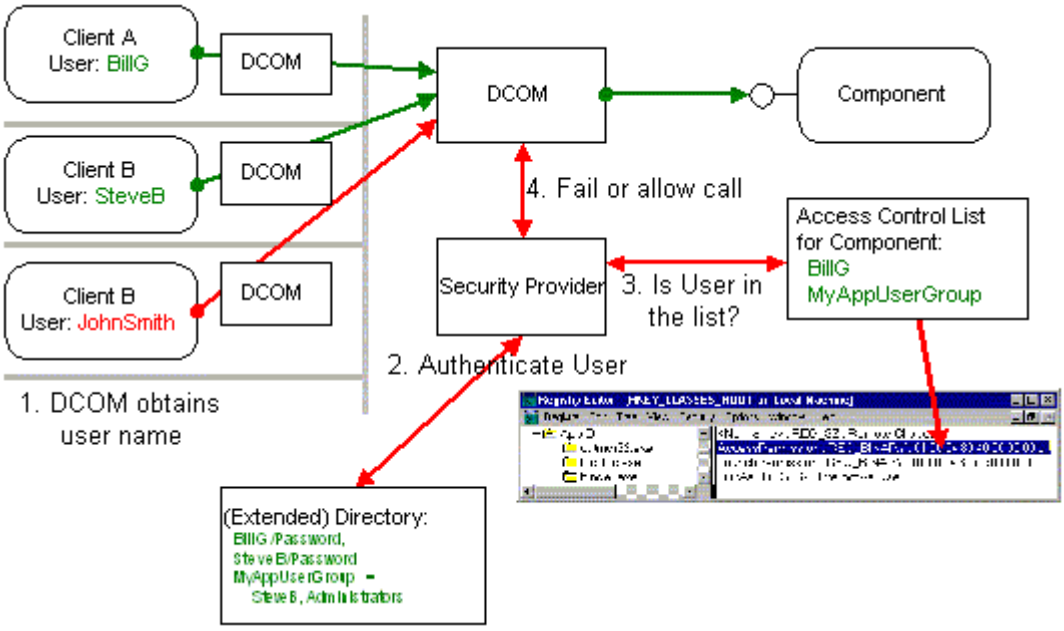

Figure 2.18 - Security by configuration

3.3.17 Load Balancing

The more successful a distributed application, the higher the load that the growing number of users places on all components of the application. Often, even the

computing power of the fastest hardware is not enough to keep up with the user demand.

An inevitable option at this point is the distribution of the load among multiple server machines. The section above, titled "Scalability," mentions briefly how DCOM facilitates different techniques of load balancing: parallel deployment, isolating critical components, and pipelining of sequential processes.

"Load balancing" is a widely used term that describes a whole set of related techniques. DCOM does not transparently provide load balancing in all its different meanings, but it does make it easy to implement different types of load balancing.

3.3.18 Fault Tolerance

Graceful fail-over and fault-tolerance are vital for mission-critical applications that require high availability. Such resilience is usually achieved through a number of hardware, operating system, and application software mechanisms.

DCOM provides basic support for fault tolerance at the protocol level. A sophisticated pinging mechanism, described below in the section titled "Shared Connection Management between Applications" detects network and client-side hardware failures. If the network recovers before the timeout interval, DCOM reestablishes connections automatically.

DCOM makes it easy to implement fault tolerance. One technique is the referral component introduced in the previous section. When clients detect the failure of a component, they reconnect to the same referral component that established the first connection. The referral component has information about which servers are no longer available and automatically provides the client with a new instance of the component running on another machine. Applications will, of course, still have to deal with error recovery at higher levels (consistency, loss of information, etc.).

With DCOM's ability to split a component into a server side and a client side, connecting and reconnecting to components, as well as consistency, can be made completely transparent to the client. Figure 2.19 is based on this idea [28].

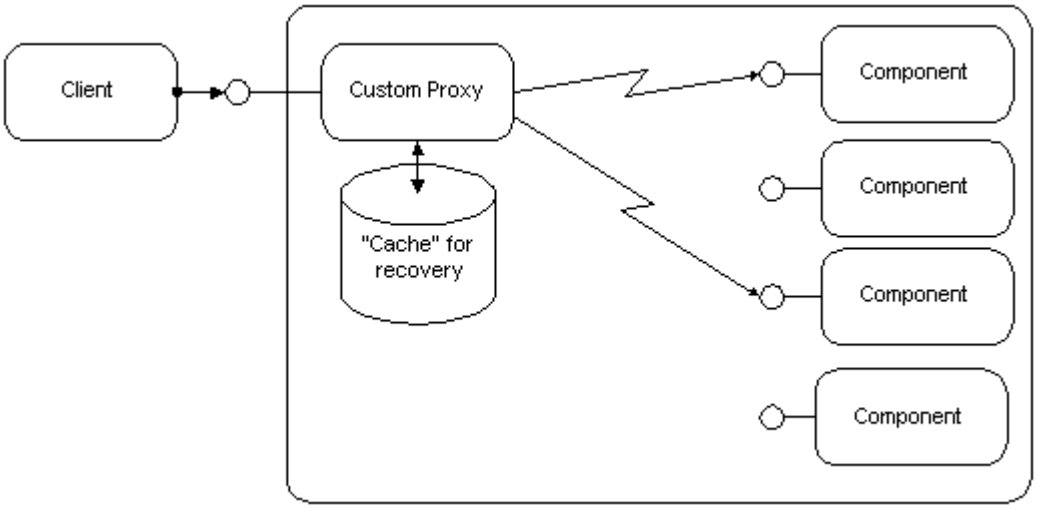

Figure 2.19 - Distributed component for fault-tolerance

Another technique is commonly referred to as "hot backup." Two copies of the same server component run in parallel on different machines, processing the same information. Clients can explicitly connect to both machines simultaneously. DCOM's "distributed components" make this action completely transparent to the client application by injecting server code on the client-side, which handles the fault-tolerance. Another approach would use a coordinating component running on a separate machine, which issues the client requests to both server components on behalf of the client.

A fail-over attempts to "migrate" a server component from one machine to the other when errors occur. This approach is used by the first release of Windows NT Clusters, but it can also be implemented at the application level. DCOM's "distributed components" make it easier to implement this functionality and shield the clients from the details.

DCOM makes implementing sophisticated fault-tolerance techniques easier. Details of the solution can be hidden from clients using DCOM's "distributed components," which run part of the component in the client process. Developers can enhance their distributed application with fault-tolerance features without changing the client component or even reconfiguring the client machine.

3.4 An overview of Remote Method Invocation (RMI) System

3.4.1 Overview of RMI

In 1995, Sun Microsystems introduced the Java programming language. Since then it has been enthusiastically adopted by software developers looking for ways to build secure, platform-independent applications for distributed network environments like the World Wide Web. Along with the advantages of its portability between platforms, Java is an attractively simple language to learn, combining many of the runtime advantages of Smalltalk and the programming concept of Modula-3 with the familiar syntax of C++. Developers can also benefit fro its remarkably complete runtime library, its designed-in security model, and its price—to most developers Java is free [1].

Increasingly complex computing environments require increasingly powerful programming techniques like object-oriented programming. The Internet enables the deployment of enormous distributed applications. Java is an object-oriented language that allows developers to produce distributed client-server applications. Distributed applications depend on the communication between “objects” residing in different locations.

RMI adds Java the power and flexibility of remote procedure calls (RPC), in any way which preserves the “object-oriented” nature of Java. It provides a framework within which Java objects in distinct Java virtual machines (JVMs) can interact. RMI is built on top of Java’s object and class facilities, its object serialization protocol, and its TCP/IP networking support [11].

The Java platform’s remote method invocation system has been specifically designed to operate in the Java application environment. The Java programming language’s RMI system assumes the homogenous environment of the Java virtual machine (JVM), and the system can therefore take advantage of the Java platform’s object model whenever possible.

3.4.2 Architecture of RMI systems

The following high-level diagrams illustrate the components that are involved in:

- A simple RMI system
- An advanced RMI system

- An RMI over IIOP system.

These examples illustrate the architecture that you will be working with when designing and building an RMI system [1, 3, 10].

Figure 2.20 illustrates the RMI system as it's simplest: a remote interface, a client, and one or more servers—remote objects—residing on a host. The client invokes

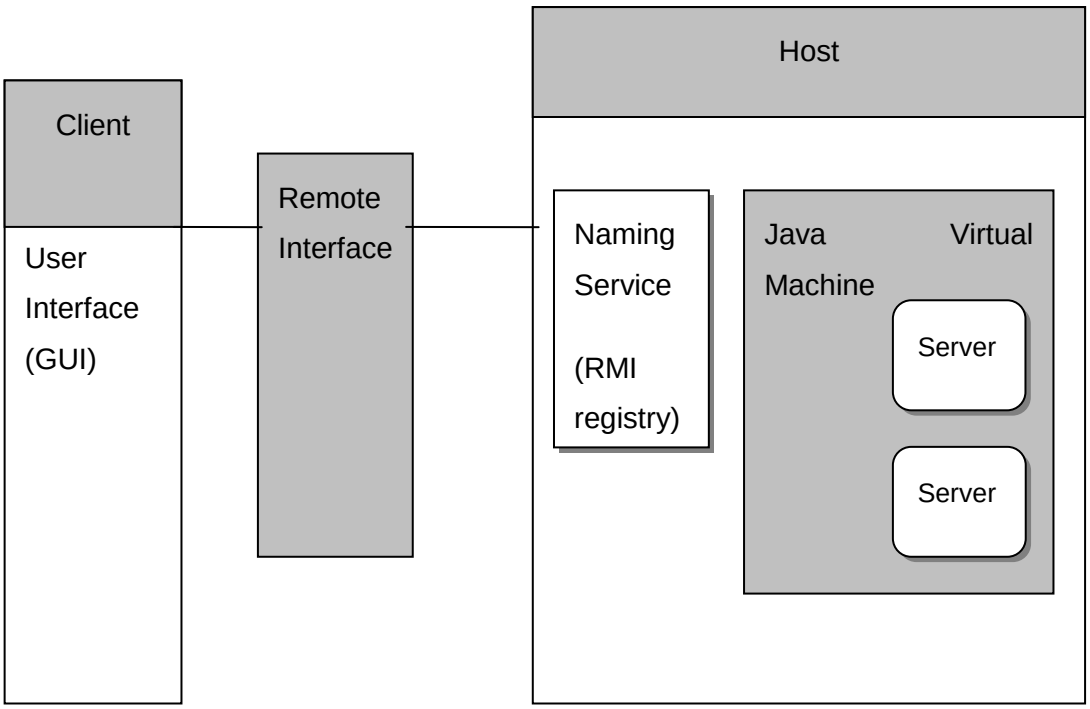

Figure 2.20 - Simple RMI System

methods on remote objects via the remote interface. A naming service such as the RMI registry resides on the host, to provide the mechanism the client uses to find one or more initial RMI servers. Figure 2.21 illustrates a more advanced RMI system, including RMI Activation—a system which allows servers to be activated on demand.

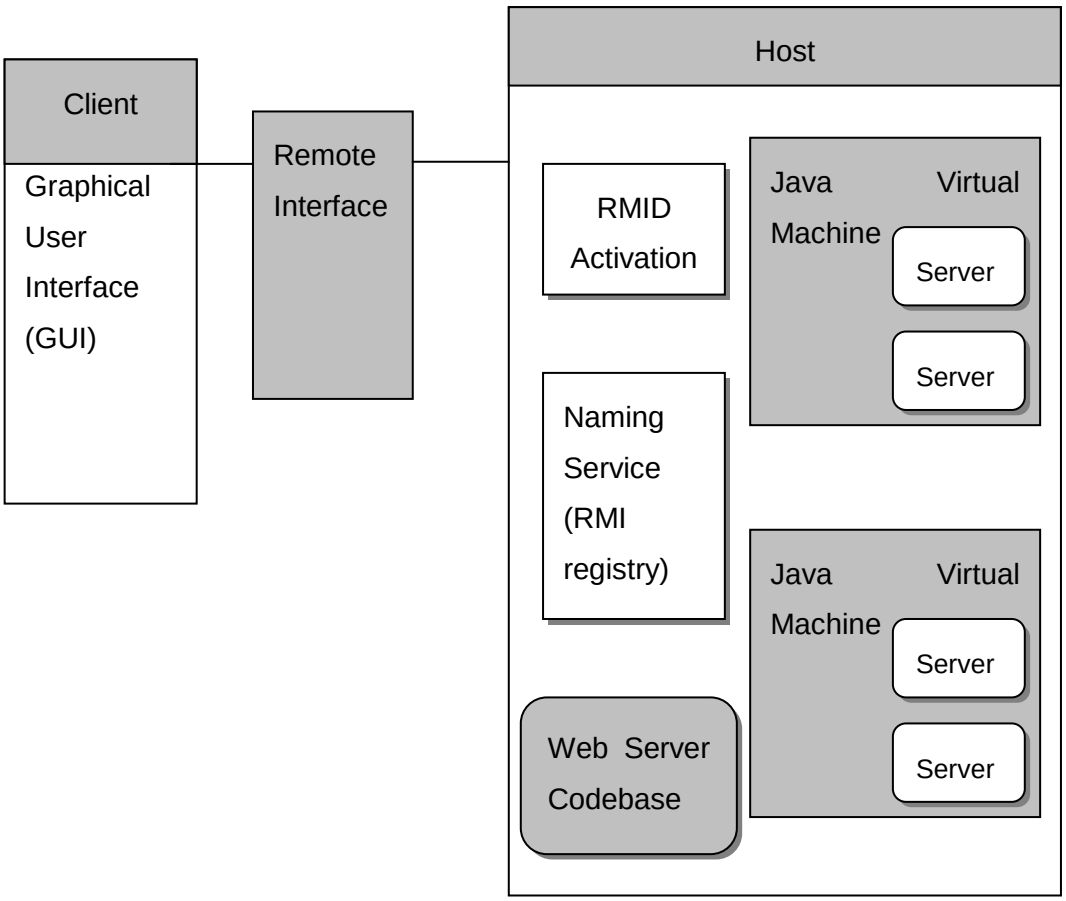

Figure 2.21 - Advanced RMI system showing codebase and activation

This diagram also illustrates that a web server providing an RMI codebase service may also be presented in an RMI system. A codebase is a global location for Java class files and Java Archive (JAR) files, accessible to RMI clients and servers.

Figure 2.22 illustrates RMI over IIOP. In RMI/IIOP, the COS Naming service, accessed via the Java Naming and Directory Interface (JNDI), is used instead of the RMI registry. In RMI over IIOP you cannot use RMI activation, which is only supported under the Java remote method protocol (JRMP).

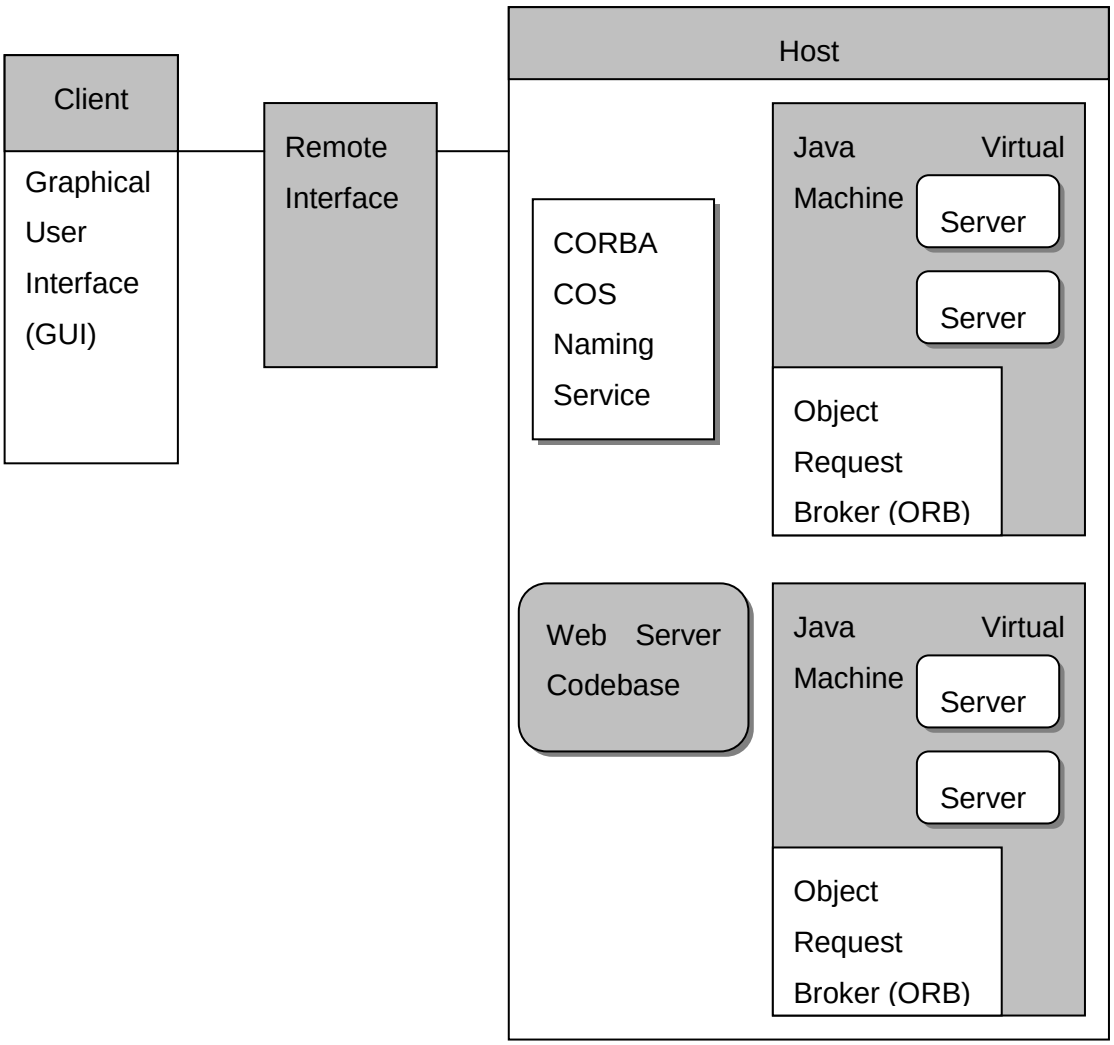

Figure 2.22 - RMI over IIOP

3.4.3 Overview of RMI Interfaces and Classes

The interfaces and classes that are responsible for specifying the remote behavior of the RMI system are defined in the java.rmi package hierarchy. The following figure 2.23 shows the relationship between several of these interfaces and classes [1]:

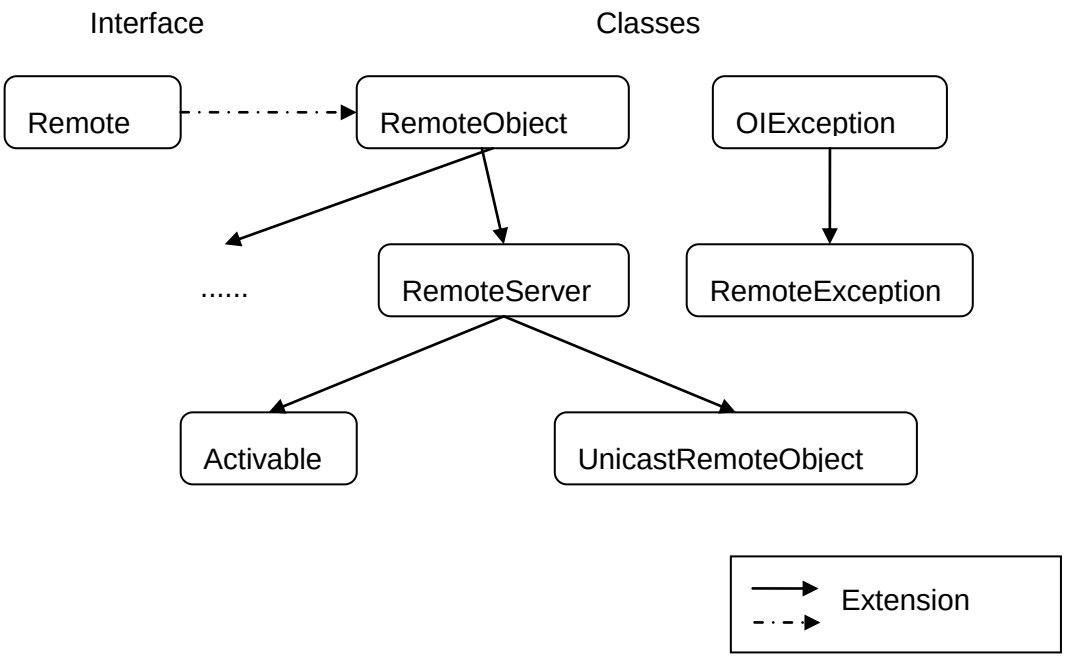

Figure 2.23 – The interface and classes

3.4.3.1 The java.rmi.RemoteInterface

In RMI, a remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine [25]. A remote interface must satisfy the following requirements:

- A remote interface must at least extend, either directly or indirectly, the interface java.rmi.Remote.
- Each method declaration in a remote interface or its super-interfaces must satisfy the requirements of a remote method declaration as follows:
- A remote method declaration must include the exception java.rmi.RemoteException (or one of its superclasses such as java.io.IOException or java.lang.Exception) in its throws clause, in addition to any application-specific exceptions (note that application specific exceptions do not have to extend java.rmi.RemoteException).
- In a remote method declaration, a remote object declared as a parameter or return value (either declared directly in the parameter list or embedded within a non-remote object in a parameter) must be declared as the remote interface, not the implementation class of that interface.

The interface java.rmi.Remote is a marker interface that defines no methods:

public interface Remote {}

A remote interface must at least extend the interface java.rmi.Remote (or another remote interface that extends java.rmi.Remote). However, a remote interface may extend a non-remote interface under the following condition:

- A remote interface may also extend another non-remote interface, as long as all of the methods (if any) of the extended interface satisfy the requirements of a remote method declaration.

3.4.3.2 The RemoteExceptionClass

The java.rmi.RemoteException class is the superclass of exceptions thrown by the RMI runtime during a remote method invocation. To ensure the robustness of applications using the RMI system, each remote method declared in a remote interface must specify java.rmi.RemoteException (or one of its superclasses such as java.io.IOException or java.lang.Exception) in its throws clause [25].

The exception java.rmi.RemoteException is thrown when a remote method invocation fails for some reason. Some reasons for remote method invocation failure include:

- Communication failure (the remote server is unreachable or is refusing connections; the connection is closed by the server, etc.)
- Failure during parameter or return value marshalling or unmarshalling
- Protocol errors

The class RemoteException is a checked exception (one that must be handled by the caller of a remote method and is checked by the compiler), not a RuntimeException.

3.4.3.3 The RemoteObjectClass and its Subclasses

RMI server functions are provided by java.rmi.server.RemoteObject and its subclasses, java.rmi.server.RemoteServer and java.rmi.server.UnicastRemoteObject and java.rmi.activation.Activatable.

- The class java.rmi.server.RemoteObject provides implementations for the java.lang.Object methods, hashCode, equals, and toString that are sensible for remote objects.

- The methods needed to create remote objects and export them (make them available to remote clients) are provided by the classes UnicastRemoteObject and Activatable. The subclass identifies the semantics of the remote reference, for example whether the server is a simple remote object or is an activatable remote object (one that executes when invoked).
- The java.rmi.server.UnicastRemoteObject class defines a singleton (unicast) remote object whose references are valid only while the server process is alive.
- The class java.rmi.activation.Activatable is an abstract class that defines an activatable remote object that starts executing when its remote methods are invoked and can shut itself down when necessary.

3.4.3.4 Implementing a Remote Interface

The general rules for a class that implements a remote interface are as follows:

- The class usually extends java.rmi.server.UnicastRemoteObject, thereby inheriting the remote behavior provided by the classes' java.rmi.server.RemoteObject and java.rmi.server.RemoteServer.
- The class can implement any number of remote interfaces.
- The class can extend another remote implementation class.
- The class can define methods that do not appear in the remote interface, but those methods can only be used locally and are not available remotely.

Note that if necessary, a class that implements a remote interface can extend some other class besides java.rmi.server.UnicastRemoteObject. However, the implementation class must then assume the responsibility for exporting the object (taken care of by the UnicastRemoteObject constructor) and for implementing (if needed) the correct remote semantics of the hashCode, equals, and toString methods inherited from the java.lang.Object class.

3.4.4 Parameter Passing in Remote Method Invocation

An argument to, or a return value from, a remote object can be any object that is serializable. This includes primitive types, remote objects, and non-remote objects that implement the java.io.Serializable interface. Classes, for parameters or return values that are not available locally are downloaded dynamically by the RMI system [26].

3.4.4.1 Passing Non-remote Objects

A non-remote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, is passed by copy; that is, the object is serialized using the object serialization mechanism of the Java platform.

So, when a non-remote object is passed as an argument or return value in a remote method invocation, the content of the non-remote object is copied before invoking the call on the remote object. When a non-remote object is returned from a remote method invocation, a new object is created in the calling virtual machine.

3.4.4.2 Passing Remote Objects

When passing an exported remote object as a parameter or return value in a remote method call, the stub for that remote object is passed instead. Remote objects that are not exported will not be replaced with a stub instance. A remote object passed as a parameter can only implement remote interfaces. Figure 2.24 explains this method.

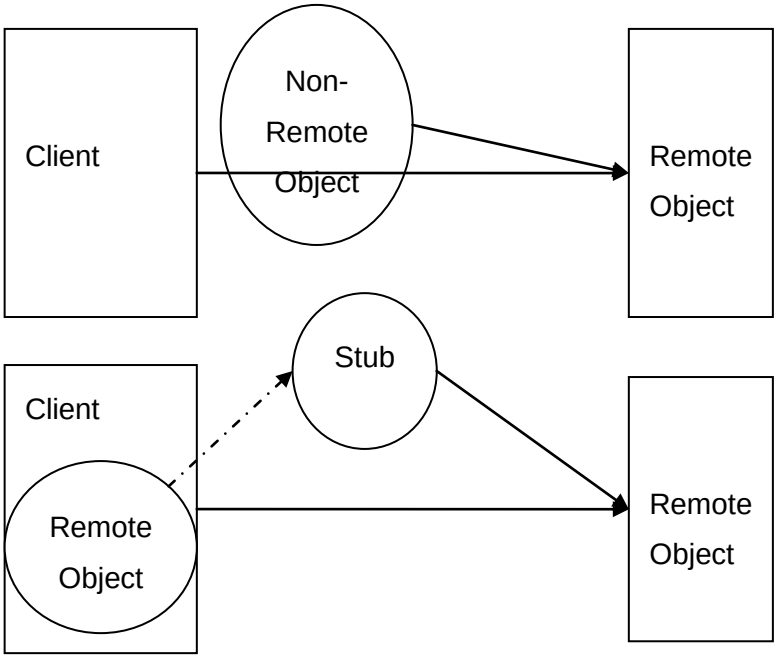

Figure 2.24 - RMI passes a stub instead of the entire object

Referential Integrity

If two references to an object are passed from one JVM to another JVM in parameters (or in the return value) in a single remote method call and those references refer to the same object in the sending JVM, those references will refer to a single copy of the object in the receiving JVM. More generally stated: within a single remote method call, the RMI system maintains referential integrity among the objects passed as parameters or as a return value in the call.

Class Annotation

When an object is sent from one JVM to another in a remote method call, the RMI system annotates the class descriptor in the call stream with information (the URL) of the class so that the class can be loaded at the receiver. It is a requirement that classes be downloaded on demand during remote method invocation.

Parameter Transmission

Parameters in an RMI call are written to a stream that is a subclass of the class java.io.ObjectOutputStream in order to serialize the parameters to the destination of the remote call. The ObjectOutputStream subclass overrides the replaceObject method to replace each exported remote object with its corresponding stub instance. Parameters that are objects are written to the stream using the ObjectOutputStream's writeObject method. The ObjectOutputStream calls the replaceObject method for each object written to the stream via the writeObject method (that includes objects referenced by those objects that are written). The replaceObject method of RMI's subclass of ObjectOutputStream returns the following:

- If the object passed to replaceObject is an instance of java.rmi.Remote and that object is exported to the RMI runtime, then it returns the stub for the remote object. If the object is an instance of java.rmi.Remote and the object is not exported to the RMI runtime, then replaceObject returns the object itself. A stub for a remote object is obtained via a call to the method java.rmi.server.RemoteObject.toStub.
- If the object passed to replaceObject is not an instance of java.rmi.Remote, then the object is simply returned. RMI's subclass of ObjectOutputStream also implements the annotateClass method that annotates the call stream with the location of the class so that it can be downloaded at the receiver.

Since parameters are written to a single ObjectOutputStream, references that refer to the same object at the caller will refer to the same copy of the object at the receiver. At the receiver, parameters are read by a single ObjectInputStream. Any other default behavior of ObjectOutputStream for writing objects (and similarly ObjectInputStream for reading objects) is maintained in parameter passing. For example, the calling of writeReplace when writing objects and readResolve when reading objects is honored by RMI's parameter marshal and unmarshal streams.

In a similar manner to parameter passing in RMI as described above, a return value (or exception) is written to a subclass of ObjectOutputStream and has the same replacement behavior as parameter transmission.

3.4.5 Locating Remote Objects

A simple bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based methods of the class java.rmi.Naming.

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is usually obtained as a parameter or return value in a method call. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts. The java.rmi.Naming class provides Uniform Resource Locator (URL) based methods to look up, bind, rebind, unbind, and list the name-object pairings maintained on a particular host and port.

3.4.6 RMI System

3.4.6.1 Stubs and Skeletons

RMI uses a standard mechanism (employed in RPC systems) for communicating with remote objects: stubs and skeletons. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements. The implementation is shown in figure 2.25 [1].

When a stub's method is invoked, it does the following:

- initiates a connection with the remote JVM containing the remote object,
- marshals (writes and transmits) the parameters to the remote JVM,
- waits for the result of the method invocation,
- unmarshals (reads) the return value or exception returned, and
- returns the value to the caller.

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote JVM, each remote object may have a corresponding skeleton (in Java 2 platform-only environments, skeletons are not required). The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshals (reads) the parameters for the remote method,
- invokes the method on the actual remote object implementation, and
- marshals (writes and transmits) the result (return value or exception) to the caller.

In the Java 2 SDK, Standard Edition, v1.2 an additional stub protocol was introduced that eliminates the need for skeletons in Java 2 platform-only environments. Instead, generic code is used to carry out the duties performed by skeletons in JDK1.1. Stubs and skeletons are generated by the rmic compiler.

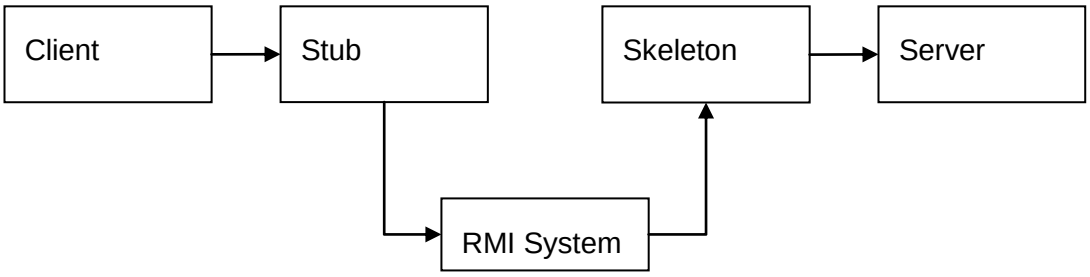

Figure 2.25 - RMI stub and skeleton basic system

3.4.6.2 Thread Usage in Remote Method Invocations

A method dispatched by the RMI runtime to a remote object implementation may or may not execute in a separate thread. The RMI runtime makes no guarantees with respect to mapping remote object invocations to threads. Since remote method

invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe.

3.4.6.3 Garbage Collection of Remote Objects

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of the remote objects' clients so that it can terminate appropriately. RMI uses a reference counting garbage collection algorithm similar to Modula-3's Network Objects. To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine, its reference count is incremented. The first reference to an object sends a "referenced" message to the server for the object. As live references are found to be unreferenced in the local virtual machine, the count is decremented. When the last reference has been discarded, an unreferenced message is sent to the server. Many subtleties exist in the protocol; most of these are related to maintaining the ordering of referenced and unreferenced messages in order to ensure that the object is not prematurely collected.

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects [1].

As long as a local reference to a remote object exists, it cannot be garbage collected and it can be passed in remote calls or returned to clients. Passing a remote object adds the identifier for the virtual machine to which it was passed to the referenced set. A remote object needing unreferenced notification must implement the java.rmi.server.Unreferenced interface. When those references no longer exist, the unreferenced method will be invoked. Unreferenced is called when the set of references is found to be empty so it might be called more than once. Remote objects are only collected when no more references, either local or remote, still exist.

Note that if a network partition exists between a client and a remote server object, it is possible that premature collection of the remote object will occur (since the transport might believe that the client crashed). Because of the possibility of premature collection, remote references cannot guarantee referential integrity; in other words, it is always possible that a remote reference may in fact not refer to an existing object. An attempt to use such a reference will generate a RemoteException which must be handled by the application.

3.4.6.4 Dynamic Class Loading

RMI allows parameters, return values and exceptions passed in RMI calls to be any object that is serializable. RMI uses the object serialization mechanism to transmit data from one virtual machine to another and also annotates the call stream with the appropriate location information so that the class definition files can be loaded at the receiver.

When parameters and return values for a remote method invocation are unmarshalled to become live objects in the receiving JVM, class definitions are required for all of the types of objects in the stream. The unmarshalling process first attempts to resolve classes by name in its local class loading context (the context class loader of the current thread). RMI also provides a facility for dynamically loading the class definitions for the actual types of objects passed as parameters and return values for remote method invocations from network locations specified by the transmitting endpoint. This includes the dynamic downloading of remote stub classes corresponding to particular remote object implementation classes (and used to contain remote references) as well as any other type that is passed by value in RMI calls, such as the subclass of a declared parameter type, that is not already available in the class loading context of the unmarshalling side.

To support dynamic class loading, the RMI runtime uses special subclasses of java.io.ObjectOutputStream and java.io.ObjectInputStream for the marshal streams that it uses for marshalling and unmarshalling RMI parameters and return values. These subclasses respectively override the annotateClass method of ObjectOutputStream and the resolveClass method of ObjectInputStream to communicate information about where to locate class files containing the definitions for classes corresponding to the class descriptors in the stream.

For every class descriptor written to an RMI marshal stream, the annotateClass method adds to the stream the result of calling java.rmi.server.RMIClassLoader.getClassAnnotation for the class object, which may be null or may be a String object representing the codebase URL path (a space-separated list of URLs) from which the remote endpoint should download the class definition file for the given class.

For every class descriptor read from an RMI marshal stream, the resolveClass method reads a single object from the stream. If the object is a String (and the value of the java.rmi.server.useCodebaseOnly property is not true), then resolveClass returns the result of calling RMIClassLoader.loadClass with the annotated String object as the first parameter and the name of the desired class in the class descriptor as the second parameter. Otherwise, resolveClass returns the result of calling RMIClassLoader.loadClass with the name of the desired class as the only parameter.

3.4.6.5 RMI through Firewalls via Proxies

The RMI transport layer normally attempts to open direct sockets to hosts on the Internet. Many intranets, however, have firewalls that do not allow this. The default RMI transport, therefore, provides two alternate HTTP-based mechanisms which enable a client behind a firewall to invoke a method on a remote object which resides outside the firewall.

As described in this section, the HTTP-based mechanism that the RMI transport layer uses for RMI calls only applies to firewalls with HTTP proxy servers [1].

How an RMI Call is packaged within the HTTP Protocol

To get outside a firewall, the transport layer embeds an RMI call within the firewall-trusted HTTP protocol. The RMI call data is sent outside as the body of an HTTP POST request, and the return information is sent back in the body of the HTTP response. The transport layer will formulate the POST request in one of two ways:

1. If the firewall proxy will forward an HTTP request directed to an arbitrary port on the host machine, then it is forwarded directly to the port on which the RMI server is listening. The default RMI transport layer on the target machine is listening with a server socket that is capable of understanding and decoding RMI calls inside POST requests.

2. If the firewall proxy will only forward HTTP requests directed to certain well-known HTTP ports, then the call is forwarded to the HTTP server listening on port 80 of the host machine, and a CGI script is executed to forward the call to the target RMI server port on the same machine.

The Default Socket Factory

The RMI transport implementation includes an extension of the class java.rmi.server.RMISocketFactory, which is the default resourceprovider for client and server sockets used to send and receive RMI calls; this default socket factory can be obtained via the java.rmi.server.RMISocketFactory.getDefaultSocketFactory method. This default socket factory creates sockets that transparently provide the firewall tunneling mechanism as follows:

- Client sockets first attempt a direct socket connection. Client sockets automatically attempt HTTP connections to hosts that cannot be contacted with a direct socket if that direct socket connection results in either a java.net.NoRouteToHostException or a java.net.UnknownHostException being thrown. If a direct socket connection results in any other exception being thrown, such as a java.net.ConnectException, an HTTP connection will not be attempted.
- Server sockets automatically detect if a newly-accepted connection is an HTTP POST request, and if so, return a socket that will expose only the body of the request to the transport and format its output as an HTTP response.

Client-side sockets, with this default behavior, are provided by the factory's java.rmi.server.RMISocketFactory.createSocket method. Serverside sockets with this default behavior are provided by the factory's java.rmi.server.RMISocketFactory.createServerSocket method.

Configuring the Client

There is no special configuration necessary to enable the client to send RMI calls through a firewall.

The client can, however, disable the packaging of RMI calls as HTTP requests by setting the java.rmi.server.disableHttp property to equal the boolean value true.

Configuring the Server

1. In order for a client outside the server host's domain to be able to invoke methods on a server's remote objects, the client must be able to find the server. To do this, the remote references that the server exports must contain the fully-qualified name of the server host.

Depending on the server's platform and network environment, this information may or may not be available to the Java virtual machine on which the server is running. If it is not available, the host's fully qualified name must be specified with the property java.rmi.server.hostname when starting the server.

2. If the server will not support RMI clients behind firewalls that can forward to arbitrary ports, use this configuration:

    a. An HTTP server is listening on port 80.

    b. A CGI script is located at the aliased URL path

    /cgi-bin/java-rmi.cgi

This script:

- Invokes the local interpreter for the Java programming language to execute a class internal to the transport layer which forwards the request to the appropriate RMI server port.
- Defines properties in the Java virtual machine with the same names and values as the CGI 1.0 defined environment variables. An example script is supplied in the RMI distribution for the Solaris and Windows 32 operating systems. Note that the script must specify the complete path to the interpreter for the Java programming language on the server machine.

3.4.6.6 Performance Issues and Limitations

Calls transmitted via HTTP requests are at least an order of magnitude slower that those sent through direct sockets, without taking proxy forwarding delays into consideration.

Because HTTP requests can only be initiated in one direction through a firewall, a client cannot export its own remote objects outside the firewall, because a host

outside the firewall cannot initiate a method invocation back on the client. So this makes the performance worse.

4. COLLABORATIVE SYSTEMS

A collaborative system is one where multiple users or agents engage in a shared activity, usually from remote locations. In the larger family of distributed applications, collaborative systems are distinguished by the fact that the agents in the system are working together towards a common goal and have a critical need to interact closely with each other: sharing information, exchanging requests with each other, and checking in with each other on their status. Collaborative system is distinguished by certain level of concurrency; i.e., the agents in the system are interacting with system and with each other at roughly the same time. So a chat session is collaborative, because all of the agents involved need coordinate with each other to be sure that the chatters do not miss anyone else's comments. An email system is not collaborative, because each email client simply wants to be sure that its message get to the right server, and eventually to the client, and does not need to coordinate with any of them in order to accomplish its goal [2, 10, 11].

Figure 3.1 depicts some of the elements that can go into a collaborative system [2]:

- Autonomous or user driven agents
- Operational and data servers
- Dynamic and persistent data repositories
- Transactions between agents, servers, and data

Basically, there are two types of collaboration between groups of people using programs, control devices, and measuring instruments.

- Synchronous collaboration occurs when all components collaborate at the same time. Teleconferencing and multi-user white boarding are examples of synchronous collaboration.
- Asynchronous collaboration occurs when components can participate at different times over the course of collaboration. An example of asynchronous collaboration is, a concurrent version-control system, with people working together on documents over extended periods of time.

There are two main approaches for the collaborative applications. One of them is collaborative-aware-applications and the other one is collaboration transparency.

Collaborative-aware-applications mean applications, which are developed for cooperative work by multiple users. In other words, many users interact with the application. In collaboration transparency, applications are developed for a single user. When the runtime environment supports collaborative transparency, it leverages the existing base of single user applications by extending them to the collaborative use. This means, all the applications can be transferred (single user applications), without any modification to the program code, collaborative programs. Because of the simplicity and power of the collaboration transparency, it is widely used for the production of the collaborative programs.

Agents, servers, data repositories, and transactions are all elements that make up distributed systems in general, but the nature of the transactions between agents and the shared goals of the agents make system collaborative [2, 9].

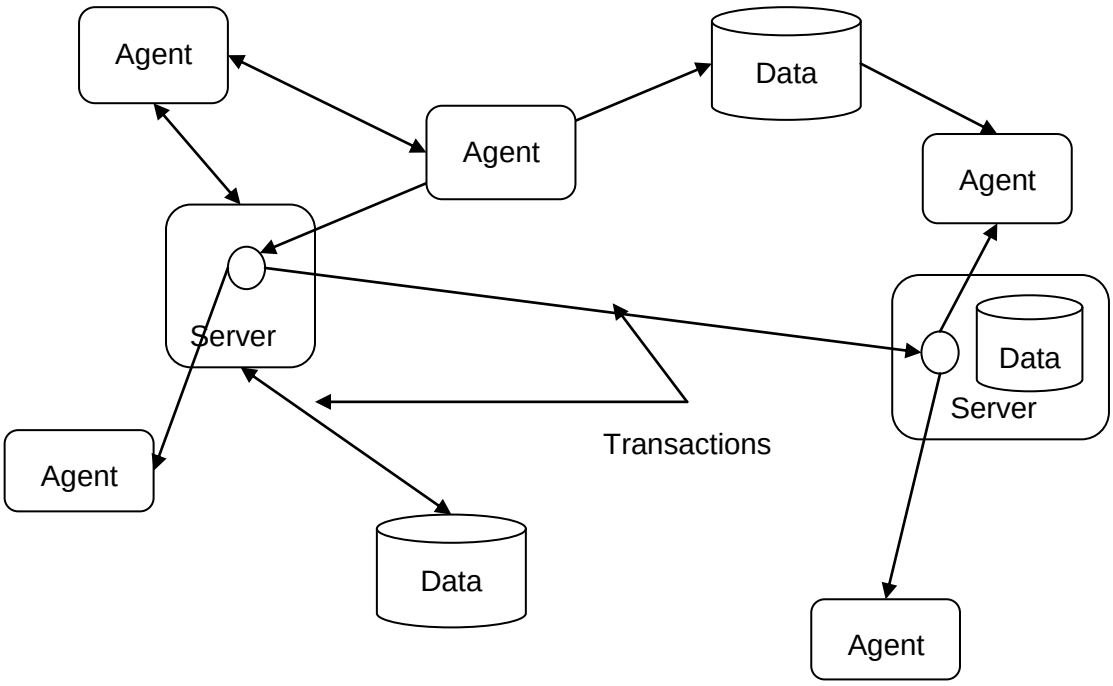

Figure 3.1 - Collaborative systems structure

Some examples of what collaborative systems are:

- Shared whiteboards
- Interactive chat
- Distributed or parallel compute engines
- Coordinated data search agents (web robots)

The first two involve collaborative agents under the direct control of human beings, while the last two involve agents that are programmed to act autonomously. So a collaborative system can involve concurrent transactions between people, between autonomous computing agents, or some mixture of the two.

4.1 Collaborative System Architecture

Collaborative systems has different needs that other distributed applications. These needs come from different nature of the system. These requirements are shown below.

4.1.1 Issues with Collaboration

For the most part, the issues that come up with collaborative systems are similar to those that arise in other concurrent programming tasks. In the context of collaborative systems, some of these issues take on even more importance; and, collaborative raises a few issues of it own.

4.1.2 Communication Needs

A collaborative system has multiple remote agents collaborating dynamically, so it must be flexible in its ability to route transactions. Depending on the application, the underlying communications might need to support point-to-point messages between agents, broadcast messages sent to the entire agent community, or “narrowcast” messages sent to specific groups of participating agents. An interactive chat server that supports chat rooms is a good example of this. Messages are normally broadcast to the entire group, but if an individual is in a single chat room, then her/him messages should only be sent to the other participants in that room. In some cases, we need to support private, one-on-one messaging between agents or users in the system [18].

In addition to simple message-like communications, there may be a need for agents to have a richer interface to other agents. Agents may need to pass object data to each other, and remote agents may need to interact directly with each other through distributed object interfaces.

4.1.3 Maintaining Agent Identities

If multiple agents are in collaboration, there has to be some way to uniquely identify them so that messages can be addressed and delivered, tasks can be assigned correctly. Also, if access to the system or to certain resources associated with the system needs to be restricted, and then participant's identities will need to be authenticated as well. Depending on the application, there may also be data or other resources associated with individual agents. This information must be maintained along with agent identities, and in some cases access to these resources must be controlled based on identities.

A practical example of this issue is a shared whiteboard application. A shared whiteboard is a virtual drawing space that multiple remote users can view and "write" on in order to share information, ideas, etc.—the digital equivalent of a group of people working around a real whiteboard in a meeting room. In order for the individuals using the whiteboard to understand what is being contributed by whom, the whiteboard system has to keep some kind of identity information for each participant. Each participant's contribution to the whiteboard must be shown with a visual indication of who is responsible for it. It may also be desirable to allow each individual the right to modify or delete only his contributions to the whiteboard, which means adding access control based on identities.

4.1.4 Shared State Information

In many collaborative systems some data and resources are shared among participants. Shared data is common to most distributed systems, but is particularly important in collaborative systems. A cooperative effort among computing agents is usually expressed in terms of a set of data that needs to be shared by all agents in the system. In our shared whiteboard example, the current contents of the whiteboard are shared among all agents. With multiple agents accessing and potentially modifying this shared state information, maintaining the integrity of the information will be an important issue. If two or more agents attempt to alter the same piece of shared state information, then there has to be a reasonable and consistent way to determine how to merge these requests, and how to make it known to the affected agents what's been done with their transactions [20].

4.1.5 Performance

Some collaborative systems have to make a trade-off between keeping shared state consistent across all the agents and maximizing the overall performance. There are situations, such as in shared whiteboard applications, where it's important that all of the agents in the system have their view of the shared state of the system kept up-to-date as closely as possible [19]. The simplest way to do this is to have a central mediator acting as a clearinghouse for agent events. The mediator gets notified whenever an agent does something that changes the shared state of the system, and the mediator is responsible for sending these updates to all of the agents in the system. This also makes it simple to ensure that updates are sequenced correctly across all agents, if that's important to the application.

The problem is that the central mediator can become a bottleneck as the size of the system scales up. If there are lots of agents to be notified or lots of changes that agents have to be notified about, then the mediator may have trouble keeping up with the traffic and the agents in the system will waste a lot of time waiting for updates. Another approach would be not to use a mediator at all, and instead have a peer-to-peer system where each agent broadcasts its update to all the other agents. While this may improve the throughput of updates, it makes it difficult to maintain consistency across the system. With each update being broadcast independently and asynchronously, it can be quite a feat to make sure that each agent ends up with the same state after all the updates have been sent, especially if the order of the updates is important.

4.1.6 The Power of the Collaboration Transparency

Many applications for single users are developed and ready for use. But unfortunately, these applications cannot be used collaboratively. The code of the applications must be written again for collaborative usage. This approach is not economical; it is time and money consuming for users and for programmers.

Applications developed for single user may be used collaboratively by modifying either the application or its runtime environment. After modification, multiple users may share the view and interact with the application. An environment that provides this application-sharing capability is called a collaboration transparency system because the shared single-user application is unaware that more than one user is interacting with it.

There are two methods, which can be used with collaboration transparency. These are:

- Display broadcasting
- Event broadcasting

Display broadcasting: The scheme of the model is shown in figure 3.2.

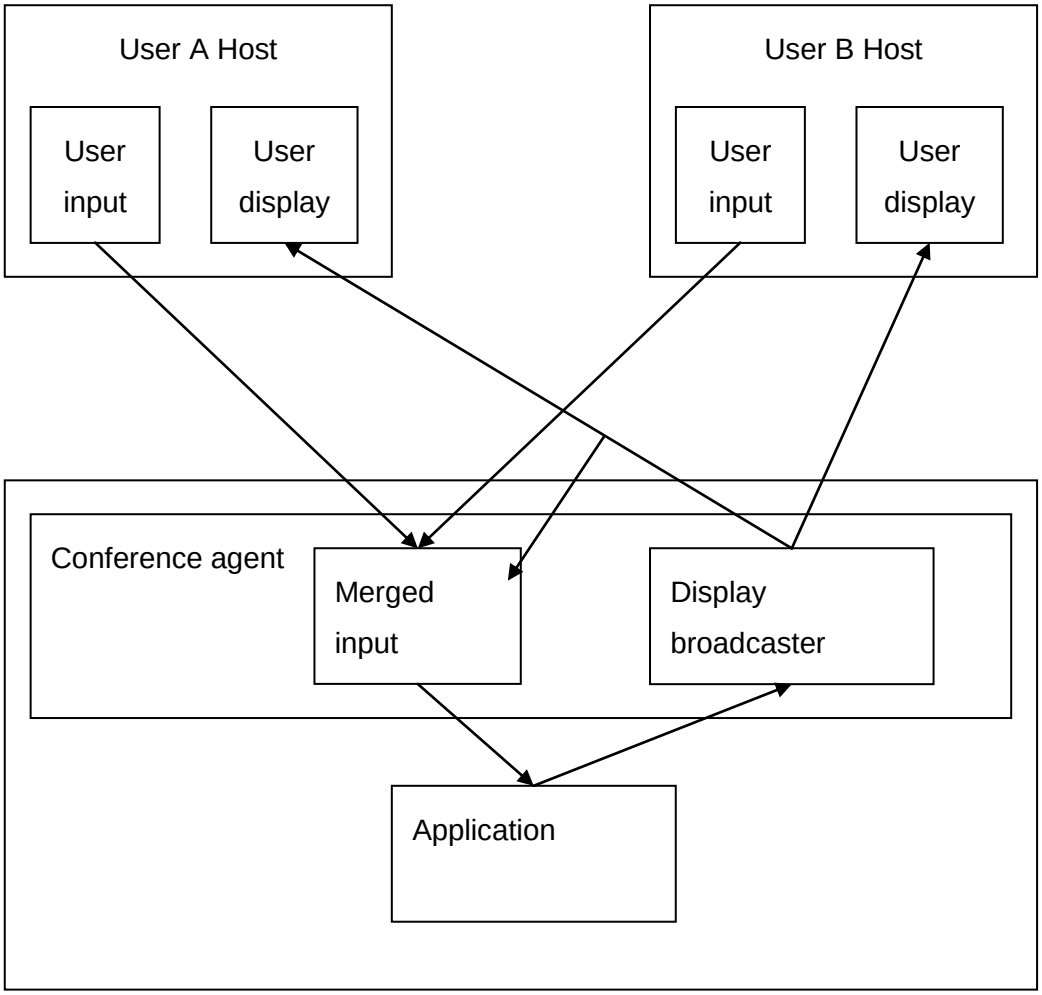

Figure 3.2 - The model of the display broadcasting

A central conference agent receives all user input and serializes the events. The conference agent then distributes display updates to the participant's windowing system [25]. This system has two disadvantages:

This system only sends the graphical user interface to the users not the original program. This is good for dumb machines but in nowadays world the machines become more and more smart. So the computational work can be done on the

client, which decreases server job and network traffic. But with this system, this cannot be done.

The conference agent must distribute large amounts of data with this approach. This can be suitable for LAN but not WAN. This is not suitable for WAN because Internet connection does not support enough speed for this kind of transportation.

Event broadcasting: In event broadcasting, each user has the copy of the application. The model is shown in figure 3.3.

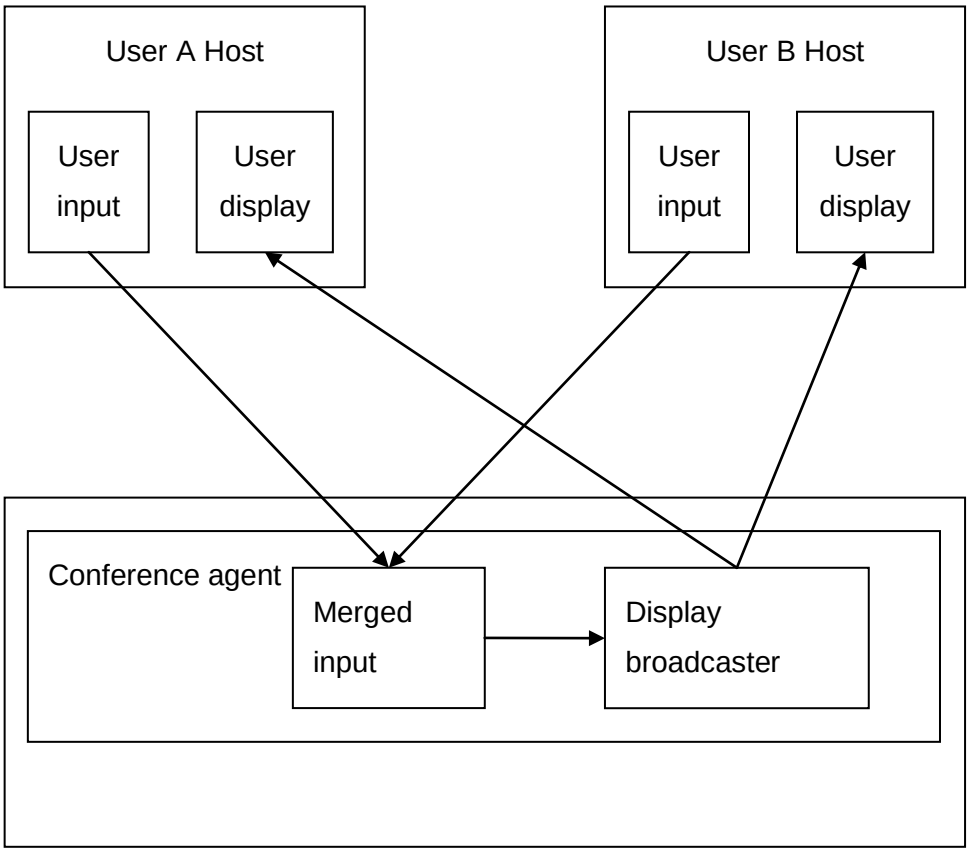

Figure 3.3 - The model of the event broadcasting

With event broadcasting, all users have the original program. When they use the program only specific event that are being generated by users are sent, not the graphical interface. So that, network traffic and server job is decreased.

4.2 Objectives and Problems of Collaborative Systems

Collaborative systems can cover different system architectures. They can be as simple as a chat system or as complicated as distributed 3D modeling system. But in common they prove same kind of job to the systems.

Collaborative systems have some extra problems than other systems. Because these applications are distributed, their problems need some extra work to be solved.

These application problems and objectives are summarized below.

4.2.1 Objectives of Collaborative Systems

Collaborative systems are generally used for communications needs between people. With using these systems; people will communicate with each other. In this communications they share different kind of materials like video, audio, presentations, graphics, 3D modeling object etc…[8, 15, 17].

Mainly, collaborative systems are used for information sharing purpose. Because this age is an information age, we can use this type of systems of every part of the life. These distributed applications are generally used for education and business.

In education, we have an environment that will provide communication between students or instructors, or both of them. This tool can be used in any part of the science because computer and science are becoming united. As an example; we can simulate a classroom even students are geographically apart. Instructor will have tools for management of class and for education. Students will have tools for learning and communication. Instructor will have and whiteboard and a video system. He/she will give lectures with video tool and may give an example with whiteboard. Student also has same tools with some restrictions. He/she will listen to instructor with video tool and watch whiteboard for examples. But students do not have permission for modifying these tools contents.

In business, we need meetings in order to direct the business. With the help of collaboration system workers will have meetings more easily and comfortable way. First of all, they do not need a meeting room and some other stuff. They can have meeting in their offices. They do not need to transfer computer related stuff to the

physical stuff. Workers also do not need to make any other additional work; like making presentations.

In both topics, collaboration system provides suitable ways for making jobs easier.

4.2.2 Main Problems with Collaborative Tools

There are some problems, which must be solved in order to have reliable and efficient collaborative programs. In these programs, system must support concurrency control and provide the necessary visual cues that online data is part of a shared interaction. Also system must preserve some separation between public and private workspace and optimize user interactivity. These topics are discussed in the next sessions.

4.2.2.1 Spontaneous Interactions

In collaborative systems, not all group interactions occur in the context of scheduled meetings. In fact, many interactions between collaborators are spontaneous and unplanned. This means that users can not always predict who they will be interacting with, nor can they anticipate which applications to be shared [9].

There will be 2 common spontaneous interactions in the system. They are about late comers and undoing any event in the system.

Collaborative system generally used for meetings. In real life we can know that there are always people who come to meeting late. Also this is the case for collaborative systems. We need to overcome with this problem. In order to bring a latecomer into shared session, shared state needs to be replicated on the newcomer's workstation. There are three basic mechanisms that are available for transferring system state:

1. The system can replay to the new latecomer the history of events that led to the current state.

2. The system can upload the shared state from an existing collaborator and download it directly into the new collaborator.

3. Process migration techniques can be applied to copy rather than move an existing collaborator.

Histories

Replaying event histories is the most straightforward approach for bringing latecomers up to date. In this approach, all events are logged by a server in the system. So when a new latecomer tries to join the system these events are sent to his/her collaborative application. This will bring the system up to date.

Direct State Transfer

A faster way to bring latecomers up to date consists of directly downloading the entire shared state information into new latecomers' application. This requires, however, that the entity being replicated support mechanism for uploading and downloading state information. Unfortunately, this brings heavy work for programmer. All collaborator applications must be aware of their state and must log all the events in the system. Also, this brings a new challenge for the floor control mechanism [12, 16].

Process Migration

Direct state transfer is not a good choice; an alternative approach consists of extending process migration facilities to allow shared applications to be duplicated on latecomers' workstations. Such facilities, however, are extremely complex. The main difficulties arise from the need to rebind connections to external services to corresponding services on the workstation of the new latecomer [16].

In summary, we can say that history mechanism is the more suitable and easiest way for supporting latecomers. Also this mechanism helps the undo operations as well.

The second spontaneous interaction is undoing any event in the system. Undo is a useful and widely supported feature which can be used to recover from erroneous operations, learn new system features, and explore alternative solutions. The ability to undo any operation at any time is especially important for collaborative editing systems because it can be used to support local or global undo and also multiple undo models.

There are not too many solutions for undo effect in collaborator systems. And many solutions are based on generally on the same principle. The framework of undo solution is shown in figure 3.4 [16].

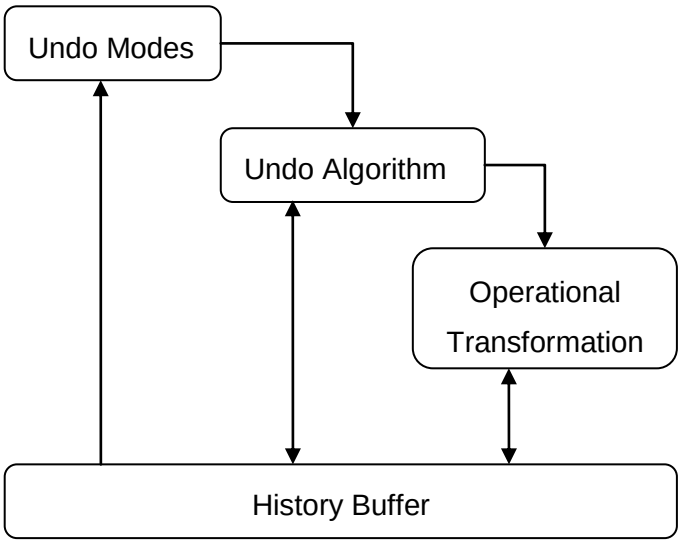

Figure 3.4 - Framework of the undo operation

This framework consists of three major components: the undo modes component, the undo algorithm component, and operational transformation component. In addition, there is a common data structure shared by all three components: the history buffer, which is used at each site to save all executed editing operations.

The undo mode component is responsible for determining which operation to be undone when the user presses the undo button from the interface. Internally, this component may work in one of the several modes selected by the user. This component should allow a user to choose any undo mode and switch from one mode to another during a collaborative editing session, and allow multiple users to use different undo modes in the same collaborative editing session. This component may use the contents in the history buffer for determining which operation to undo and for providing undo interface information.

The undo algorithm component is responsible for carrying out the necessary steps to achieve the effect of undoing the selected operation. This component determines what inverse operation should be generated and executed in order to eliminate the effect of the original operation selected for undo, and how to represent and treat the inverse operation in the history buffer after its execution. The essential requirement for this component is that it must have the capability to undo any operation at any time. This component may update the history buffer in carrying out the steps involved in the undo algorithm.

The operational transformation component provides transformation service to the undo algorithm component. It transforms the inverse operation into a proper form for execution and transforms the operations in the history buffer to accommodate the impact of the inverse operation.

4.2.2.2 Floor Control and Access Control Mechanisms

Floor control and access control mechanisms are about the usage of the collaboration system. With this controls, we can use the system more effectively and reliable. With floor control, we have the right to make some job in the system [12, 13].

Within shared workspace, floor policies are employed to control access to the shared workspace. Each system must decide the level of simultaneity to support and a granularity at which to enforce access control. This will prevent system chaos when all users try to do something. With access control mechanism, every user has some rights to do something in the system. Some will have more rights than some others. This will decrease the usage of system resources and prove system security. With this tool we have a hierarchical system which is more effective and efficient.

Floor Control

The simplest floor control mechanism is, letting only one user having the floor control. Other users must wait floor until the one who holds, releases it. To obtain the floor, one may be required to take an explicit action, like pushing a special key, whereas less restrictive systems allow any keyboard or mouse activity to signal a floor change.

Floor control is an extremely contentious area of debate, with policies ranging from permitting only one person at a time to control the entire shared workspace, to running “open floor” with anyone generating input at any time to any window. These policies can be characterized along three dimensions:

1. The number of floors-one for the entire conference, one per shared application, or one per window.

2. The number of people who can “hold” a floor at the same time.

3. How the floor is passed (or handed off) between floor holders. This includes the potential use of auxiliary communication channels, especially audio.

The arguments over the first two dimensions amount to arguments over the amount of concurrent activity to be permitted. Separate floors correspond to sub-conferences and number of holders per floor corresponds to how the meeting is being run (“floor control” in its most restricted sense).

With respect to how the floor is passed, this is where technology can clearly be used to improve upon traditional social protocols. For example, the system can maintain lists of people who have asked for the floor, and can optionally introduce hysterics by selecting from this list at random rather than from the head of the queue. Or, when the basic algorithm is preemptive-the requester gets the floor on demand-the system can ensure that a request for the floor is not granted unless the current floor holder has been inactive for some time.

All arguments considered, the only certainty is that no one policy will suffice for all groups, in all situations. For example, one geographically distributed team of programmers has reported the need to alternate between open floors when “brainstorming” to one-person-at-a-time when wanting to make sure a particular piece of code is coded correctly. In general, running open floor becomes increasingly difficult as the number of participants increases, as anyone who has participated in a large meeting (especially a multi-party audio conference) can attest.

Moreover, experience indicates that even a shared window system tailored to a particular group’s behavior will require different mechanisms (and policies) depending on the operating environment in which a conference takes place. In particular, running open floor becomes increasingly difficult as communications delay increases. Anyone who has engaged in a telephone conversation over a satellite link can appreciate the types of conflicts that arise.

Finally, the availability of audio and video links may significantly influence the floor control mechanisms required in the conference agent. In particular, if a high-quality audio link is available, and is synchronized with the data channel, there may not be a need for rigid floor handoff policies, since conference participants can use traditional social protocols over the audio link to negotiate access to the shared workspace. Indeed, rigid floor control mechanisms might even obstruct these negotiations. However, as suggested in our allusions to satellite communications and multi-party audio conferences, access to the audio link can itself be a problem-the solution to which may well be mechanisms in the conference agent.

The ideal shared window system provides mechanisms that supports for a broad

range of policies indicated above, in architecture capable of switching between different policies depending on user preferences and operating environment. Moreover, in support of running open floor, the ideal system would provide mechanisms to enable recovery from conflicting input-that is, to prevent or compensate for data corruption resulting from simultaneous input.

Access Control

Access control is an indispensable part of any information sharing system. Collaborative environments introduce new requirements for access control, which cannot be met by using existing models developed for non-collaborative domains. In general a generic access control model for collaborative environments should support: [13, 14]

- Multiple, dynamic user roles: The model should allow users' access rights to be inferred from their roles. Moreover, it should allow users to take multiple roles simultaneously and change these roles dynamically during different phases of collaboration.

- Collaboration rights: Besides traditional operations such as read and write, all other operations whose results can affect multiple users should be protected by collaboration rights.

- Flexibility: The system should support fine-grained subjects, objects, and access rights, that is, it should allow independent specification of each access right of each user on each object. For instance, it should allow independent protection of each line in a multi-user code viewer.

- Easy specification: Users should be able to specify access definitions easily.

- Efficient storage and evaluation: The storage of access definitions and evaluation of the access checking rule should be efficient.

- Automation: The model should make it easy to implement access control in multi-user applications.

With these rules we can develop a collaborative system that covers the needs of our system. According the system that we develop we can combine the floor control and access control. Or we can distinctly define these controls.

4.2.2.3 GUI Enhancements

There are two types of method for GUI enhancement. These are:

- What-You-See-Is-What-I-See (WYSIWIS)
- What-You-See-Is-Not-What-I-See (WYSINWIS)

WYSIWIS method enforces a strict policy fro the display. But, a simple mapping of an application's GUI from single-user mode to a group-oriented mode is not always effective (e.g., multiple cursors on the screen may lead confusion)

WYSINWIS method on the other hand, allows a mixture of public and private windows, and personalized windows layouts.

4.2.2.4 Workspace Architectures

There is ongoing debate about the optimal underlying architectures for computer based shared workspaces. The architectural choices are classified as centralized, replicated or hybrid.

With the centralized approach, as illustrated in figure 3.5, user only has the window or GUI not the program. When a change occurs server sends all the data about GUI to all participants. In replicated architecture, which is shown in figure 3.5, each user has the copy of the program. So, server only needs to send event data not all the data about GUI. This method is more flexible and comfortable. Hybrid approach mixes the best of these schemes.

Whether a centralized, replicated or hybrid approach is chosen, there are other difficulties that arise in large communication environment like the Internet.

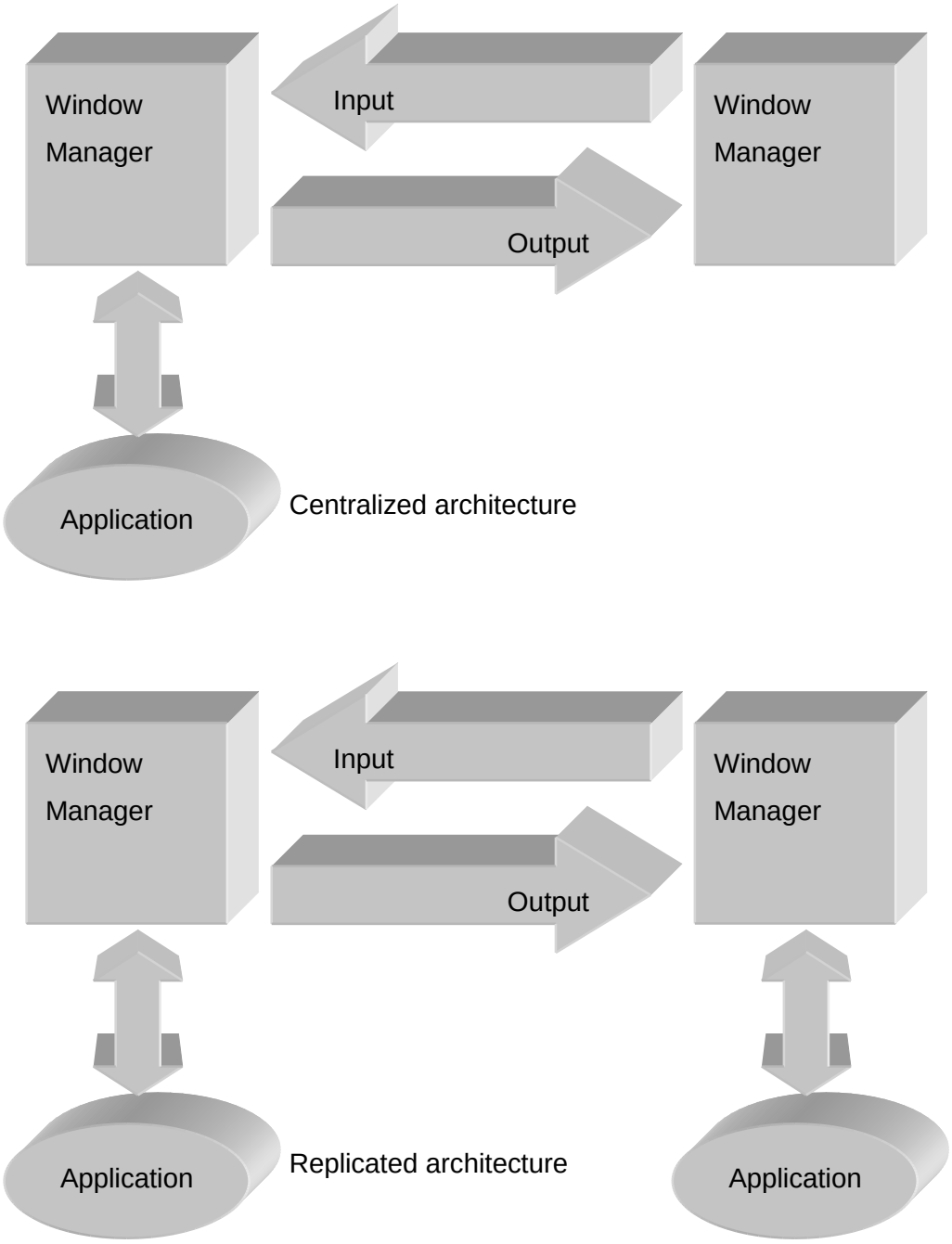

Figure 3.5 - Workspace architectures

4.3 Examples of Collaborative Applications

There are many applications that focus on providing a software infrastructure for collaboration applications. Examples of such applications are distributed whiteboards, calendars, and editors where multiple users collaborate towards one goal. Software systems that support collaborative applications include the E

programming language, Caltech Infosphere project, Java Collaborator Toolset, Habanero, Knitting Factory, Corona Upper Atmospheric Research Collaboratory (UARC), Java Remote Method Invocation, and TANGO. In this section three applications are compared. These are TANGO, Habanero, and Corona.

4.3.1 TANGO

TANGO is an integration platform which enables implementation of Web-based collaborative environments. The system provides means for fast integration of Web and non-Web-applications into one multi-user collaborative system [23].

4.3.1.1 Mechanism

Tango is a modest prototype of an open, extensible system that provides a technological framework for building subsequent generations of collaborative systems. It is a functional model that addresses and embraces a number of unresolved issues in the field of Computer Supported Collaborative Work. Its goal is to create shareable information spaces.

In TANGO, different applications can exchange messages and act accordingly. The applications exchanging information can reside on either the same node or on different machines. it allows us to build complex applications from small, reusable modules. Consider, as an example, a tandem of a chat and a web browser. This tandem is replicated on multiple nodes. As the users chat, they may pass information about interesting URLs in the chat window. In TANGO system, this information will be automatically recognized and passed to the set of browsers working in the collaborative mode. TANGO is very tightly integrated with the Web. In the functional sense, tight integration with Web implies access to the entire informational potential of the Web.

TANGO provides support for all synchronous collaborator functions, including session management, data/event distribution, flexible floor control, and multiple, configurable security levels. The system does not impose any restrictions on a number of concurrent sessions, users, or collaborative applications. Synchronous collaboration is supported via database back-end. Session record and playback capability has been designed into the system as its fundamental component. This capability applies to both events and data streams and can be used review collaborative sessions in asynchronous fashion. Playback/review capability is

provided also for real-time continuous data streams such as audio and video. TANGO provides scaleable multimedia support. Recognizing different performance requirements of continuous multimedia streams, TANGO provides mechanisms for decentralized distribution of such streams under TANGO session control. While almost entire TANGO runtime and most of the applications are written in Java, there is no restriction on the language used to implement collaborative applications compatible with TANGO.

4.3.1.2 Functionality

The main functionality provided by the system consists of the following elements:

1. Session management: A session is a group of application instances currently working together in the collaborative mode.
    a. Creating a session
    b. Joining an existing session.
    c. Leaving a session
2. Communication: Only applications belonging to the same session can communicate. System provides means for this communication. It is implemented on the basis of message passing. If an application sends a message, this message will be delivered to all other compatible applications in this particular session.
    a. Control messages
    b. Application messages
3. User authentication and authorization: System provides means of user authentication and authorization.
4. Event logging

4.3.1.3 Architecture

1. System elements: The system consists of the following components:
    a. Local daemon

b. Central server

c. Local Applications

d. Java applets

e. Control application

2. Event flow

3. TANGOsim extension

4. Video teleconferencing

4.3.1.4 Implementation

1. Daemon implementation: Daemon provides a mechanism for Netscape components such as Java applets, JavaScript scripts and plug-ins to talk to each other. The daemon has been implemented as a plug-in.

2. Server implementation: The central server was implemented as a standalone Java application.

3. Control application: Control application has been implemented as a Java applet.

4. TANGO API: TANGO API is used to port a standalone application into a shared application that can be collaboratively run under the TANGO system.

4.3.2 Habanero

NCSA Habanero is an environment that enables creation of the collaborative environments. System provides session management tools, so that the users may start and joins sessions as well as retrieve information about other participants. Habanero also supports shared state and shared actions that are distributed to all the clients or to the selected group of clients [22].

Habanero is implemented using centralized—client-server architecture. The server routes all the messages between applications, transfers shared state and actions, provides arbitration and synchronization mechanisms. The state of the tool is sent by marshalling and unmarshalling. Those operations are responsible for serialization

i.e. encoding and decoding of the shared objects. Encoded objects are sent using the channels of communication provided by the server and decoded by the receivers. Using entity called arbitrator, which resides on the server, some limits on the user actions may be imposed. Several types of the arbitrators are supported. The arbitrators manage locks that define the types of actions permitted to execute by the users.

4.3.2.1 Mechanism

Habanero is a framework for sharing Java objects with colleagues distributed around the Internet. Included, or planned, are all the networking facilities, routing, arbitration and synchronization mechanisms necessary to accomplish the sharing of state data and key events between collaborator's copies of a software tool.

Authentication and privacy features are also planned. There is no inherent limit in the number of tools per session, nor is there a limit on the type of tools that may be shared. As the project progresses, additional capabilities will enable routing of Habanero session information to a very wide number of participants. A limited, but representative set of applications is provided. No security model is presently in place.

4.3.2.2 Functionality

1. Enables other software developers to easily transform single-user applications into multi-user, shared applications.
2. Session control:
    a. Start session
    b. Join an existing session
3. Get information on the other session participants
4. Provides direct access to the collaborative applications

4.3.2.3 Architecture

The architecture of the system is based on server and client parts.

4.3.2.4 Implementation

NCSA Habanero TM is a new version of Groupware-based applications. The Habanero object-sharing framework is written entirely in Sun Microsystems's Java (TM) language.

4.3.3 Corona

Corona is a communication service developed as a part of the Upper Atmospheric Research Collaboratory (UARC) project. Corona supports group collaboration. Its main purposes are to provide space scientists with tools to effectively view and analyze data gathered by various remote instruments installed in Greenland and to create environment for collaboration between distributed communities of scientists [21].

Corona provides services based on two models:

1. publish/subscribe
2. peer group

Publish/subscribe service enables entity called publisher to send data to large number of recipients (subscribers). This is basically anonymous form of communication. The publishers know about all the subscribers but subscribers do not know about each other. Such model is preferred for distribution of the instrument readouts. In peer-group model all group members are aware of each other. They may exchange data with each other with some limitations. Limitations are imposed in form of different roles: observers and principals. The observers may only receive data while principals may send and receive data. This model is appropriate for the collaborative work of small groups of scientists. The system provides failure detection. If the confirmation does not come, messages are retransmitted and if it does not have desired effect crashed participant is removed.

4.3.3.1 Mechanism

To discover systematic and detailed approaches to the design of effective team-science support systems. These approaches are based on the use of user-centered object-oriented methods for analysis, design and constructions and the use of deployments of prototypes for real use as a means of design verification.

4.3.3.2 Functionality

1. Info: Shows the application name, version number, authors, copyright information and help.

2. Edit: This is a standard NeXT application item, and its sub items are the usual editing operations Cut, Copy, Paste, and Select All.

3. Suggestions/Bugs: opens up a window that contains an electronic mail message to be sent to the UARC developers. This is a convenient way to give the developers feedback, suggestions for changes, and bug reports.

4. Who's logged On: Opens up a status box that shows that is currently logged on. This window stays open until the user closes it. As long as the window is open, the list of users currently logged on is continuously updated to show the current set of users.

5. What's Available: Presents a small window that displays a list of the instruments that are currently generating data.

6. AllSky Camera: Displays the all sky camera data and sends command to the all sky camera.

7. Fabry Perot: Display options available for the Fabry Perot interferometer: Intensity, Neutral Temperature, Meridional Winds, Zonal Winds, Zenith Winds, and Signal.

8. IRIS: Display imaging riometer images and magnetometer data as a function of time; gives a north-south cross section of the riometer data and an east-west cross section of the riometer data.

9. Sondrestrom Radar: Provide the current radar status, reflect the current mode of operation and position of the radar, and display the current radar parameter.

10. Annotations: Allows one to access the annotation.

11. Messages: Allow one to open two message windows, in which all communications among current users of the UARC application are displayed.

12. Windows: Allow one to manipulate any of the windows that are currently open.

13. Hide: This allows one to hide UARC. All the application windows are made invisible and the menus of the application are removed from the screen. The application continues to execute and update the application displays.

4.3.3.3 Architecture

UARC 5.1 is the client end of a system of servers and clients. The total set of servers and clients consists of a connected set of instruments, instrument servers, a data transport server, an annotation server, a shared window server, and a number of clients. The path of data from the instrument to the clients display is as follows. Data is packaged into a "quick look" format at a computer (usually an IBM clone) attached to the instrument itself. This instrument computer is usually connected, via Ethernet, by TCP/IP sockets to a data transport machine, where data is packaged for the UARC system, into an internal representation, a data "object." These data objects are then broadcast to all the clients, where they are "opened" and the data is formatted and displayed. Instrument data flows from the instrument, to a central routing server, to the client, to the client display. Collaboration among clients is achieved by a flow of data from client to client. The system is based on a set of interacting objects.

4.3.3.4 Implementation

The system adapts Client-Server Model. Each of the data streams originating at the instruments is routed through a data server which, in turn, supplies the data to the user application. The user application is a client which requests data and other services from the server. The client-server model was also used for the annotation and shared window services.

5. THE INFRASTRUCTURE OF CHAT AND WHITEBOARD SYSTEM

The shared whiteboard consists of a single mediator (server) handling interactions among multiple collaborators (clients). Each collaborator has a unique identity, issued by the mediator, and each collaborator can either broadcast messages to all of the collaborators registered with the mediator, or it can send message to a single collaborator.

One of the first steps in developing a collaborative system is deciding what kind of communications scheme is right for the application. There are several ways to connect remote agents, including basic socket communications, message passing, RMI remote objects, and CORBA remote objects [2].

5.1 Chat and Whiteboard System

Chat and whiteboard system is a human managed collaborative system. The main target of the system is, providing communication to collaborators based on graphical and written basis. System only needs a web browser and an internet connection for working properly. System does not need any other application to be build or some extra system changes. With the help of the collaboration system, people who are geographically apart can have interactive communication with reliability and efficiency. The figure 3.6 shows the general working principles of the system.

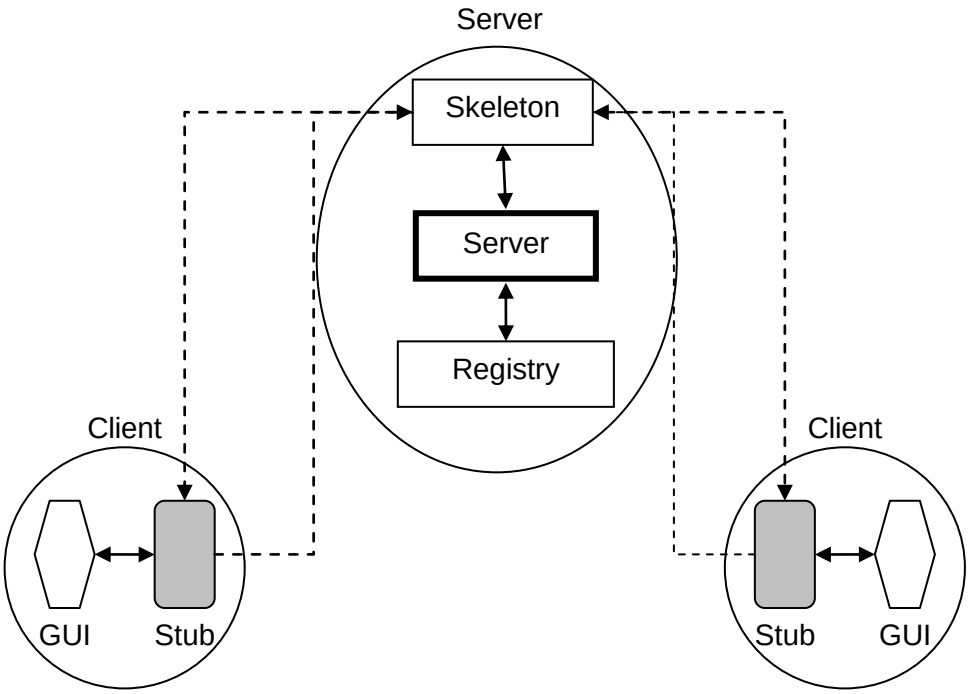

Figure 3.6 - The general system architecture

In the design phase of the project, we try to implement an application, which helps both message and drawing passing between each user. So each user can communicate each other, as well as they can draw anything on the board to share their drawings with the other users. This kind of application will help people to share more than words in the Internet. Also this implementation is flexible to supply more functions (like video, audio, picture sharing). People can also share files in this program. These features make this program completely collaborative tool, which provides strong and efficient interaction between the users. The general scheme of the project is shown below in figure 3.7.

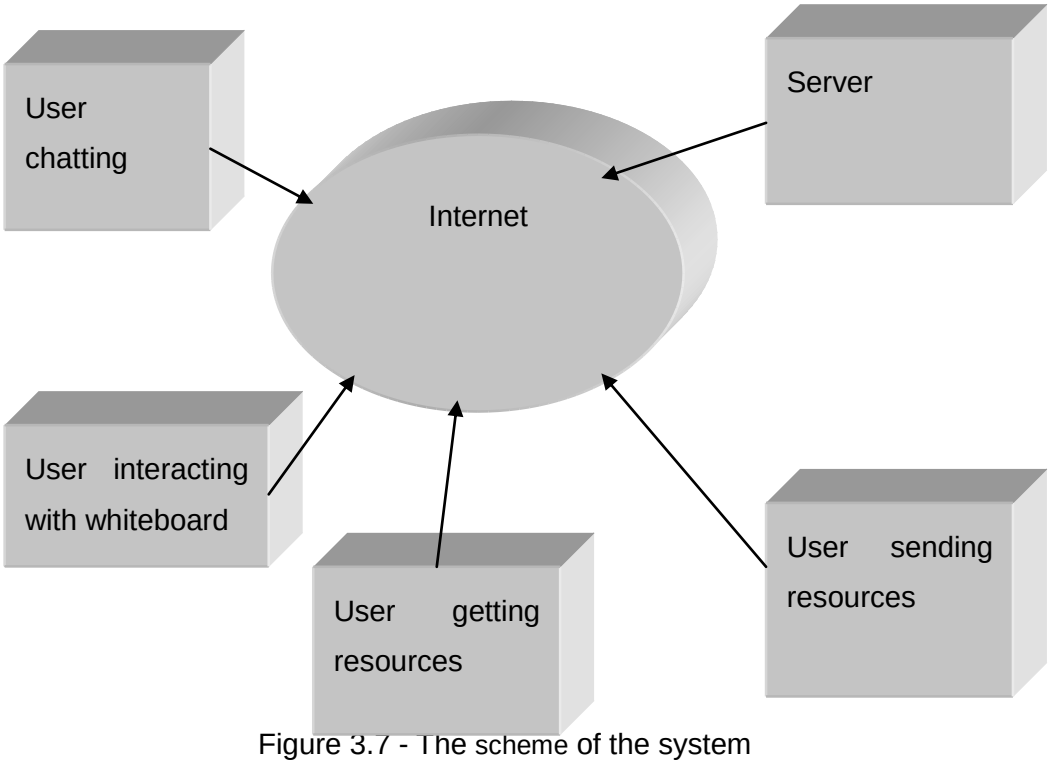

Figure 3.7 - The scheme of the system

This is the most general scheme of our project. This system is implemented in RMI. So this project has an RMI system as a background process, which is not visible to the user. This system is shown below in figure 3.8.

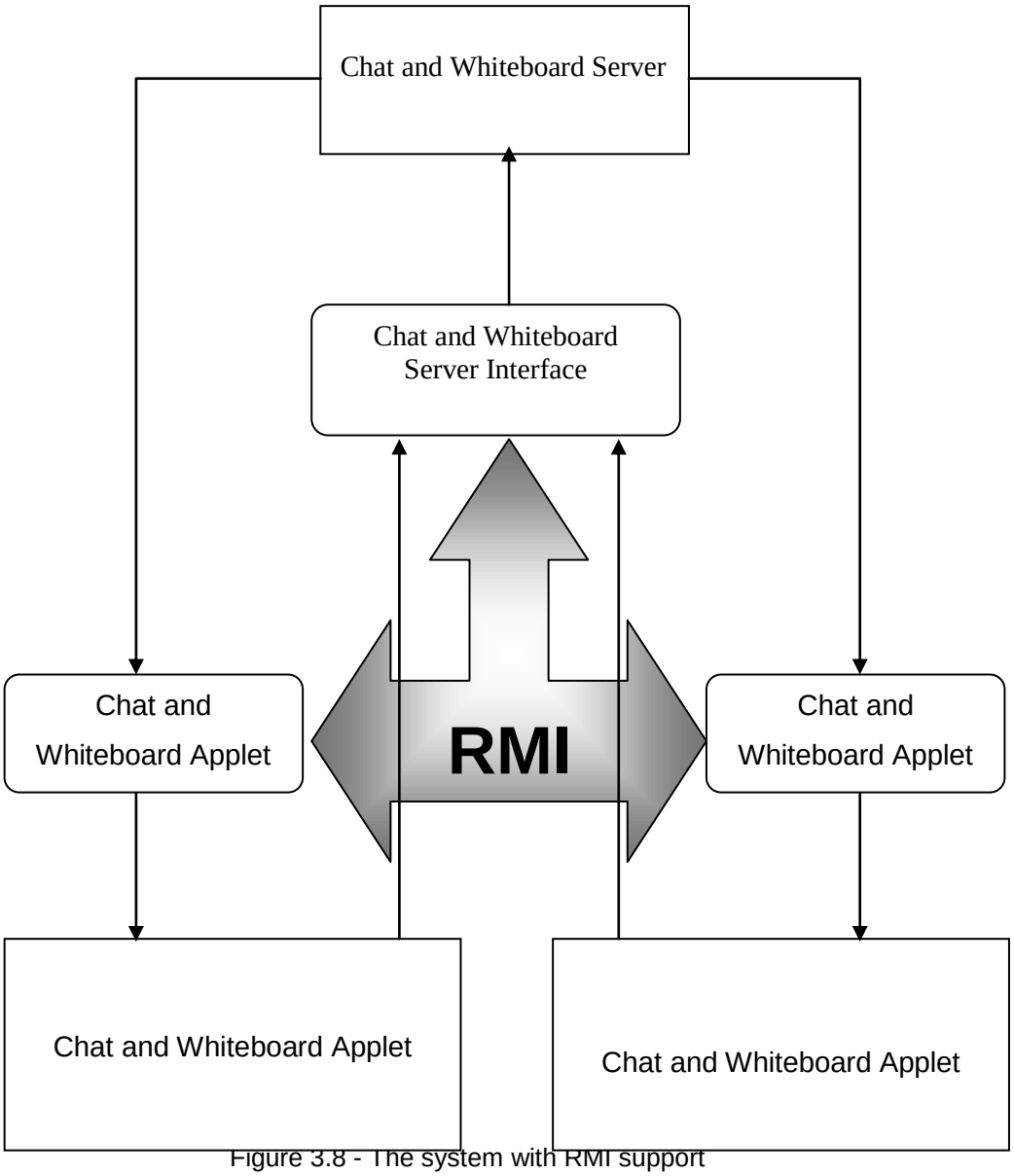

Figure 3.8 - The system with RMI support

The chat and whiteboard system can be used for several purposes. One of them is for using as a virtual classroom in education system. A virtual classroom can be build with this system which is not necessary that students and lecturer must be in the same place. In addition, because the system events are captured, students can reach the lecture stuff afterwards. Another usage can be in business environment. The meeting in business can be done easily with this tool. This tool brings the comfort and easiness to the meetings. With the help of the chat and whiteboard system brainstorming can be done easily.

5.1.1 The Architecture of Client

Client is written as an applet which application cannot harm the user' computer. So that it can be reached and used anywhere; where Internet is available. Client firstly attempts to locate a registry service which server application remote interfaces are loaded. After that, client registers itself to the registry. Finally, client displays the user interface to the participants. At the same time client also is registered to the server.

At the client side, every user inputs which take actions on the system are processed by the client interface. They are packed as an object and send to the server side. And also server side messages are taken by the client side. They are processed and necessary objects are unpacked. After these work which are done on thread structure, these messages are displayed on the graphical user interface. There are two important components that belong to client application. First one's class structure is shown in figure 3.9. This class is created by the server for each client. It provides a communication channel between the client and the server.

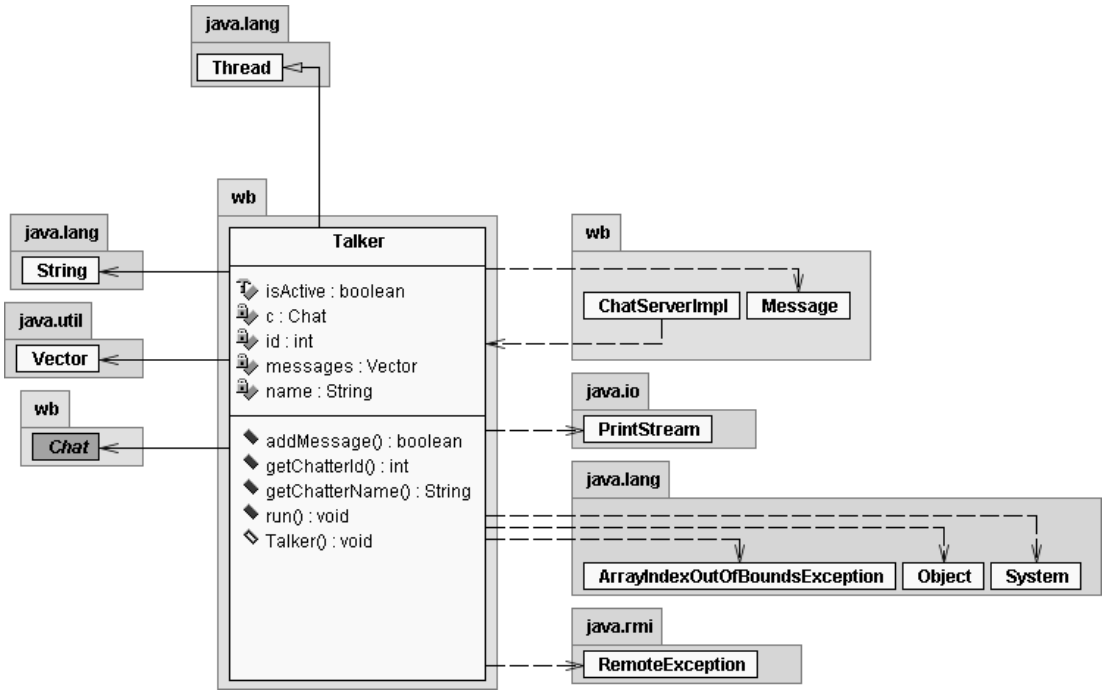

Figure 3.9 - Class structures of Talker object

The other class is the interface between user interactions with the user display and with the server. It displays the user actions as well as sending them to other users by the server. Also this class structure is shown in figure 3.10.

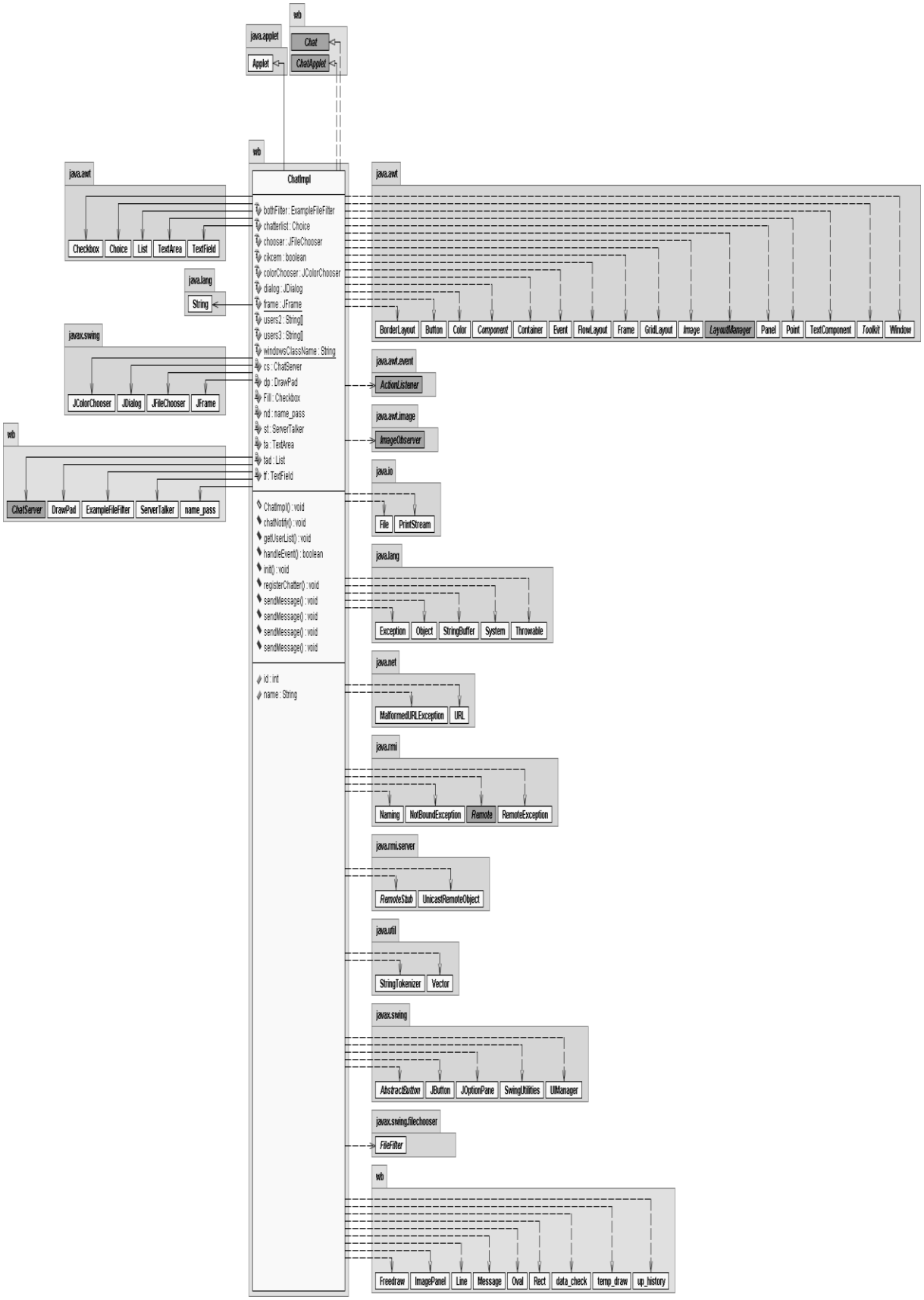

Figure 3.10 - The class file of Chat object

Another important implementation factor on the client side is, enabling remote methods, like server enables to clients, for server. These remote methods can be used only by server which client is registered. With this mechanism, we have more simple and robust system. The client application classes' are summarized in table 3.1.

Table 3.1 - Components of the Chat and Whiteboard Client

ChatImpl	This is the implementation part of the client application.
ChatApplet	This is the interface that declares the remote methods available on the server application.
Chat	This is another interface for client application
Talker	This is the thread object that is responsible of the client which is registered to server.

5.1.2 The Architecture of Server

Server enables clients to register themselves and maintains a table active of active users. When a user wants to send message to the other users, a message is sent to the server application and is then distributed to the other users. This is also true for drawings.

Server firstly starts a registry service and registers itself. Secondly, it exports method to enable clients to register themselves. Other important initialization is about the login, undo and history mechanisms. All the user events are recorded to databases in order to provide undo and history. When a client tries to login to system, his/her username and password are checked for authorization. After these operations, client and server are ready to be used. When a client enters a message or draws something on the board, it is sent to the server and then recorded to database. When server takes these inputs, it distributes all of them to the registered users. Server uses simple Message object, which enables to send a message or line of drawing.

For authentication and undo mechanisms, a database structure is used. In the database system, three tables are built. One is used for holding user information.

The other ones are used for holding user events. For database operations stored procedures are used. The structures of the tables are shown in figure 3.11 below.

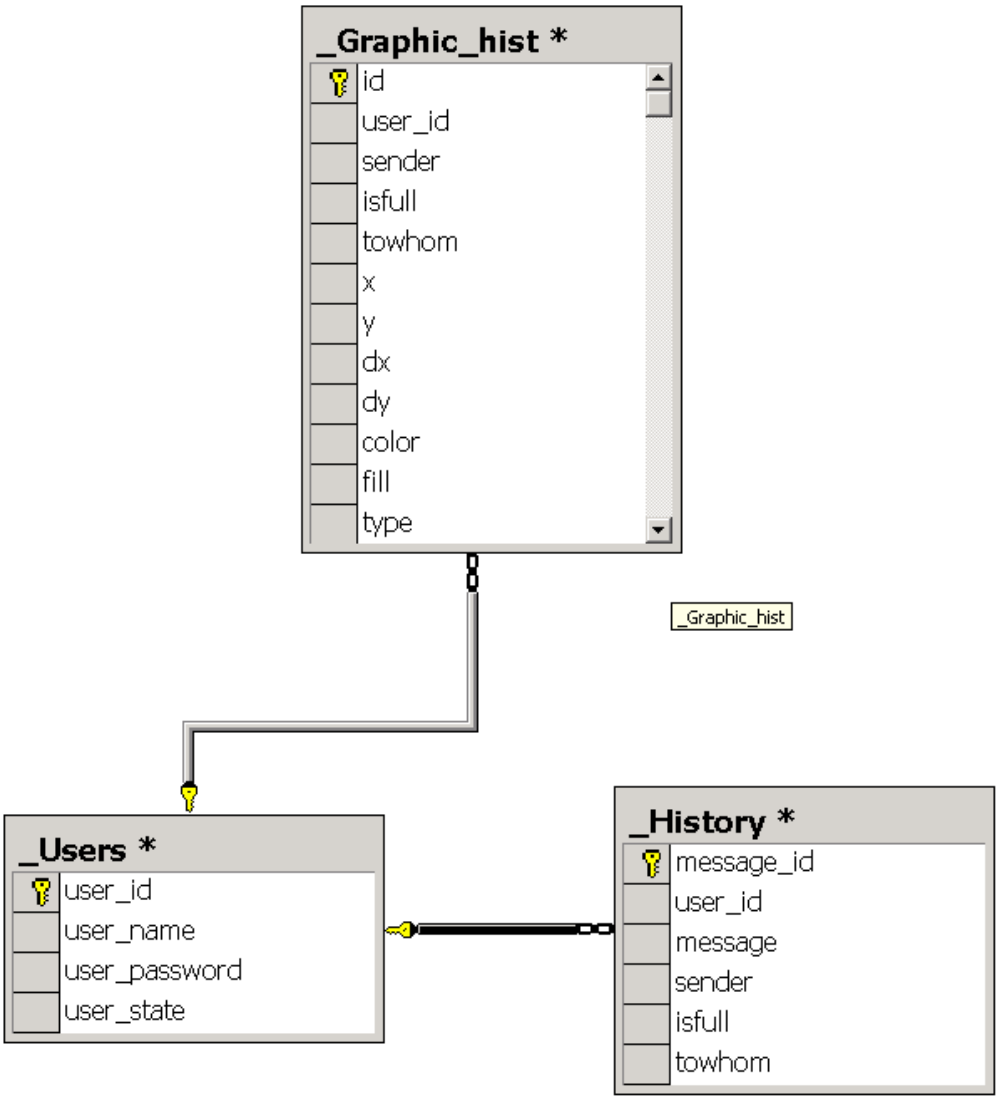

Figure 3.11 – Table Structure of the system

This database system is working on server. So that we do not concern about client security and client connection with the database. Free drawing will take too much space in the database system. Because of that, database system does not record free drawing. Database and server system must be on the same network in order to function correctly.

Chat and whiteboard server has many class structures.

These are shown in table 3.2 below.

Table 3.2 - Components of the Chat and Whiteboard Server

ChatServer	This is the interface part of the server, which describes the remote methods.
ChatServerImpl	The implementation of the server. This code generates stub and skeleton parts.
ServerTalker	This is the class that is constructed by each client in order to use remote methods in the server.
Up_history	This class is used for capturing user events in the system for history and undo mechanism.
Db_check	This class is used for authentication of each collaborator which is trying to register itself to the system.

5.1.3 Graphical User Interface

The main designing principle of the chat and whiteboard is making the system simple, useful and robust. So while designing the graphical user interface, we pay more attention to these topics. Most components of the graphical user interface are shown in figure 3.12.

<table>
<tr><td colspan="2">Application Buttons</td></tr>
<tr><td>Chat and System Messages</td><td>Drawing Panel</td></tr>
<tr><td colspan="2">Message Box for Sending Messages</td></tr>
<tr><td colspan="2">The area that will display all the collaborators’ events</td></tr>
</table>

Figure 3.12 - Graphical User Interface

As seen in the figure, the system consists of five main parts. The application buttons part is used for drawing objects (rectangle, oval), color picking, and image selection. Also to clear the messages and the drawing panel we use the application buttons. Undoing any action in the system and getting the history is also supplied by these buttons. At the left side of the graphical user interface, the system and user messages can be read. The message box at the bottom is used for sending messages to other users.

One of the main classes in graphical user interface is which is holding the drawn objects in the system. Except free drawing, they have same infrastructure. While drawing these objects, only the collaborator who draws the actual drawing object can see the object. Other collaborator can see the object after the drawing has finished. In free drawing all the instances of free drawing is sent all the users at the same time. The class structure is shown in figure 3.13.

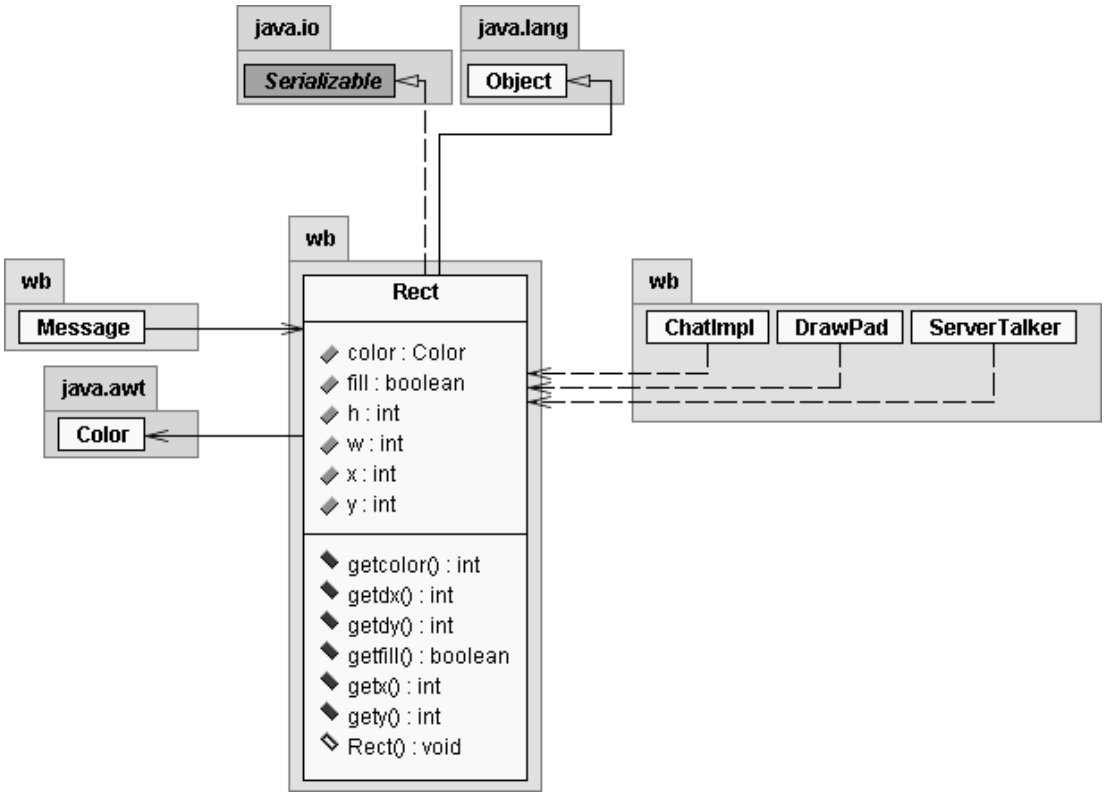

Figure 3.13 - The class structure of rectangle object

Another object in graphical user interface is image. Image object will upload the given image to the server and sends its server address to all users. With this object any collaborator can display images in his/her computer to other collaborators.

Another object which is responsible sending text message is called message object. It holds the message text, sender, and receiver.

Draw panel is also an important object which is responsible for displaying drawn object in the system. The actual user input, and other users' input is displayed on it. It is the object that represents the drawing object on the panel. The main class structure of this object is shown in figure 3.14.

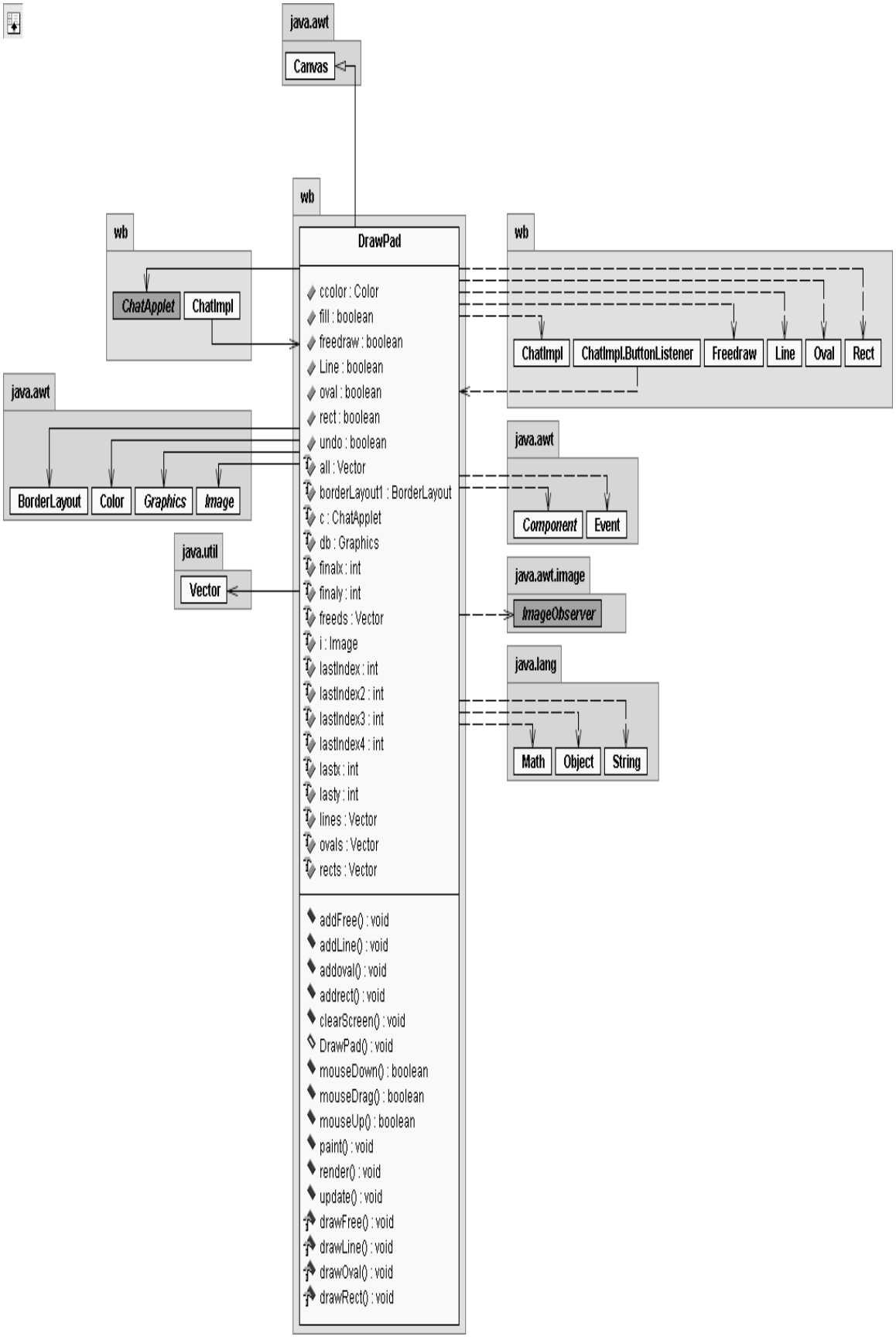

Figure 3.14 - The class structure of draw panel

The graphical user interface objects are summarized in table 3.3

Table 3.3 - Components of graphical user interface

FileFilter	This is an object for filtering and selecting image files from the operating system.
Message	An object, which describes the messages. Messages can be text or line (same as server component).
NameDialog	A simple dialog for entering the nickname of the user.
Oval	This is the object that represents an oval segment
Line	This is the object that represents a line segment (same as server component).
Rect	This is the object that represents a rectangle segment
Freedraw	This is the object that represents free drawing.
Temp_draw	This is used as a general structure for drawn objects in the system.

5.1.4 The Comparison of Chat and Whiteboard System with Existing Systems

Although many collaboration tools are implemented, their implementation mechanisms are different from one to another. Many of them does not use any distribution system technologies like CORBA or RMI. They implement their own system, and used it for their collaboration tools.

Chat and whiteboard system is implemented on RMI system. The mechanism which isolates network jobs is provided by RMI system. Because of RMI is implemented a general framework for all distributed system, some extra work must be done as compared with other system. In section 3.3, three systems are examined and compared. In this section chat and whiteboard application will also examined in this way. This will prove a pure comparison of all systems.

Chat and whiteboard system is a system that enables interactive communication between collaborators. It has textual and graphical tools. With this tool collaborators can communicate with each other. This system is a base application which will be adapted to a specific application easily. We can modify this application as a virtual classroom or as a virtual meeting system in business environment.

5.1.4.1 Mechanism

Chat and whiteboard has a framework that will enable a virtual communication platform for people. The system is server centric. Each user must register themselves to the server. And the connection between user and database is also handled by the server.

Because system is fully implemented with object oriented methods, it is easy to use this application as a framework for other collaboration applications.

5.1.4.2 Functionality

1. RMI Registry: This tool is used by server for registering its remote methods and itself to the public. And this tool is also used by clients for accessing remote objects.

2. DB Mechanisms: It provides structure for undo and history mechanisms.

3. User login: When user starts the application, a user login window displayed. User must enter the correct values for username and password.

4. Buttons: Each button on client application is self explanatory. They are told in graphical user interface section 3.5.3 widely. As a summary they provide tools to use the system.

5. DrawPad: This component is used for drawing purposes. Every drawn object is displayed by this component.

6. Text Box: This box is used for textual communications.

5.1.4.3 Architecture

Chat and whiteboard application is a three tier application. Its main structure consists of one database, one server, one RMI registry, and one or more clients. The server must be run first. It registers itself to the RMI registry. If the RMI registry is not working it initialized it. After this operation, server makes the necessary operations with the database. Client registers itself to the server by RMI registry. While registration process, also database operations are done. When client leaves the communication session, necessary operations are taken by the server.

5.1.4.4 Implementation

Server and client are implemented with thread mechanisms. Server creates threads for each individual user. Also client application creates thread for communication needs with server. So all system is a multithreading system that will prevent crashes and enables speed. The flow diagrams of server and client is given below.

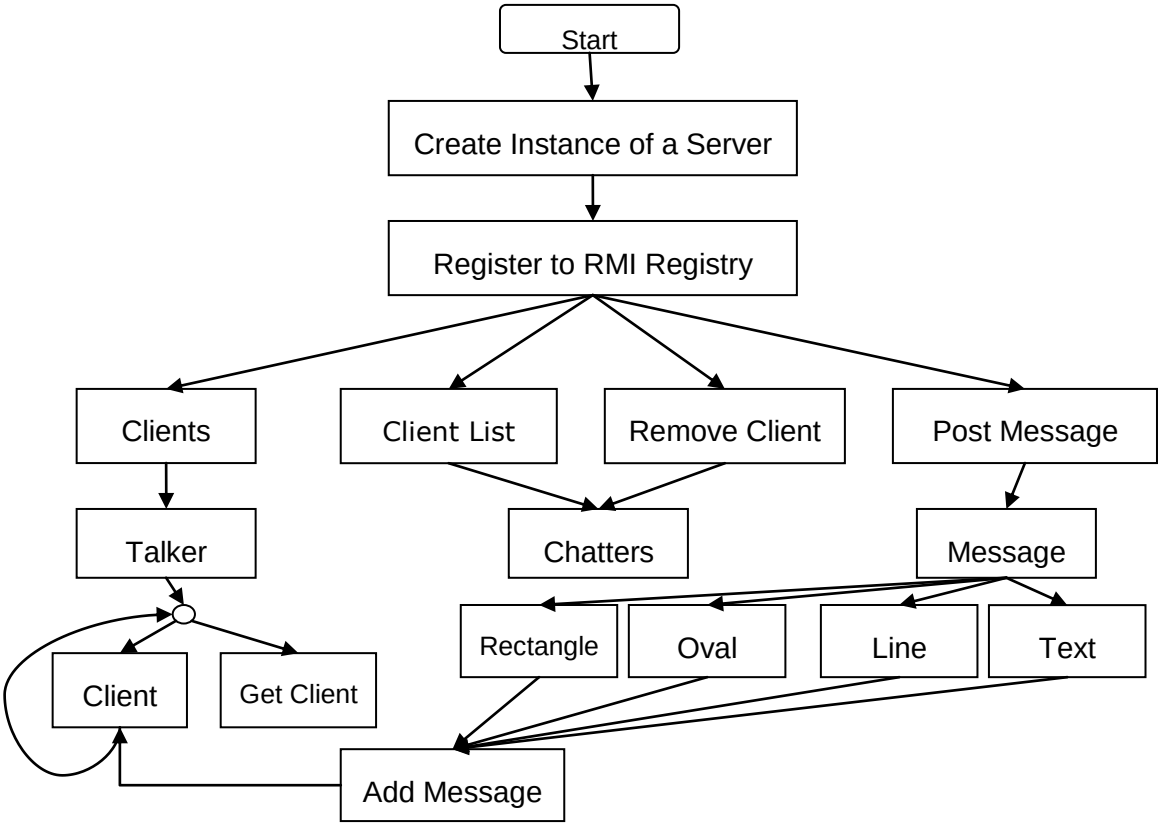

Figure 3.15 - Flowchart of server application

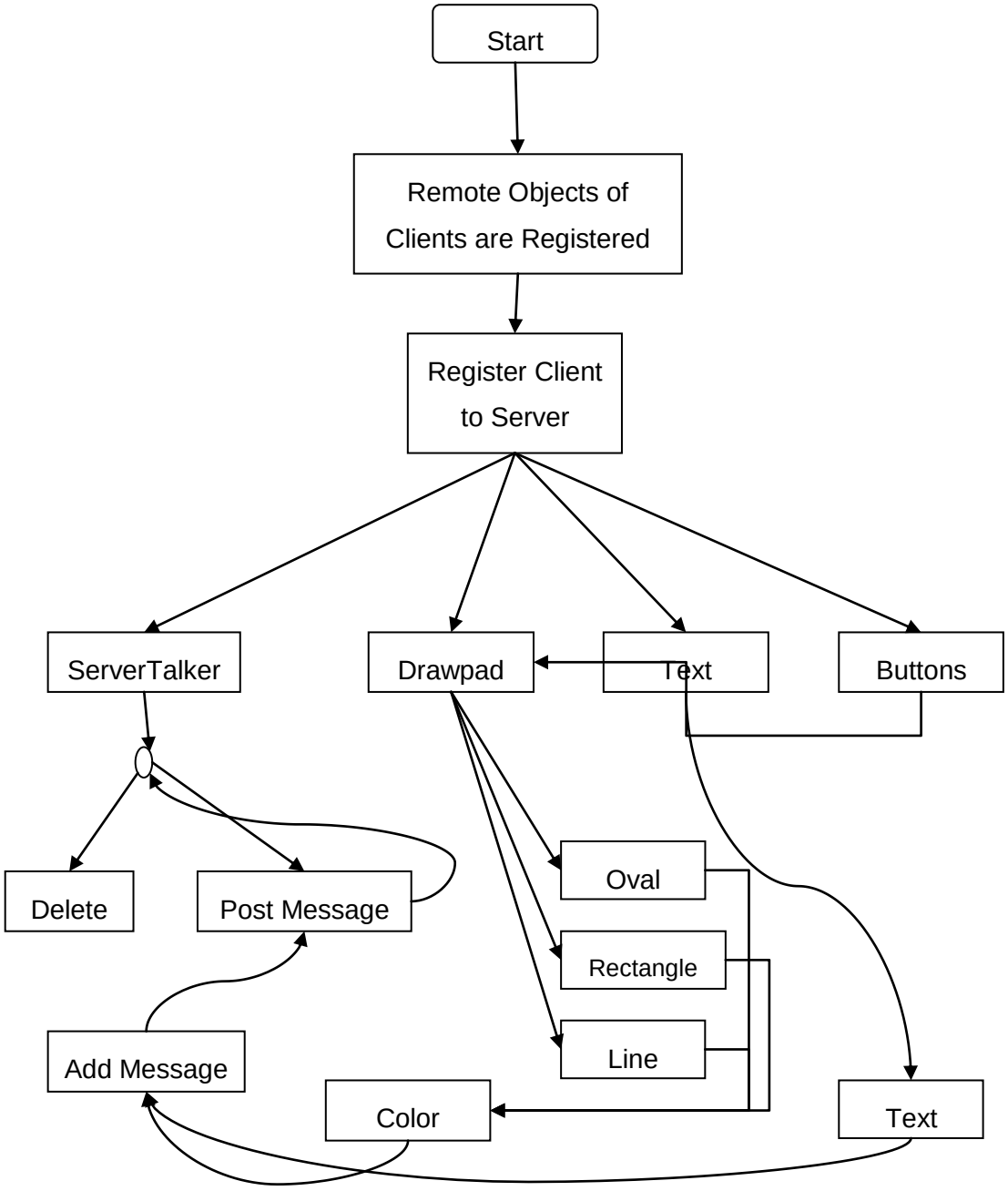

Figure 3.16 - The flowchart of client application

6. CONCLUSION

In this part, the advantages and disadvantages of systems which are being used in the project will be discussed. First of all, distributed systems will be presented. Secondly, advantages and disadvantages of RMI system will be discussed. And at last, some issues that will be solved in the future implementations are discussed.

6.1 Distributed System

Developers are engaged in a major Internet-oriented thrust to build client-server applications based on distributed object models using various implementations of RMI, CORBA and DCOM. The move away from large mainframe applications is explained by the advantages offered by distributed object technologies. Distributed object application development enables:

6.1.1 Code Re-use

As with object-oriented programming, a major benefit of distributed object development is the ability to re-use code. Code reutilization makes RAD a reality, empowering programmers to quickly and efficiently develop highly functional applications using existing plug-and-play components. Code re-use is also cost-effective, since developers are not required to reinvent the wheel for each instance of a client-server application or new piece of functionality. Finally, as a developer reincorporates components from a production environment, code re-use diminishes the amount of required system testing since these mature components have previously undergone extensive evaluation and testing.

6.1.2 Isolated Development

Distributed object applications allow for isolated development and modification of components because of the approach's modular design. Distributed object systems are broken down into separate, self-contained modules that can be worked on independently of each other, yet are capable of inter-operating with other

components. For inter-operation to occur, the modules must communicate using a common protocol or interface. But since the methods and functions existing within a module are isolated, they may be separately developed to meet system requirements without worrying about ramifications on code in other parts of the system. This feature facilitates large-scale projects by enabling multiple teams to work synchronously on distinct modules of an application, then pull the modules together via the agreed upon protocol or interface.

6.1.3 Code Maintenance

Networked applications are extremely maintainable when developed using a distributed object approach. Once again, due to the modular design of distributed object applications, changing the functionality of an application to fix known problems or meet new requirements -- a common occurrence in today's constantly changing business world -- does not require a complete application overhaul. Instead, the modules that need modification are carefully identified and then the required code changes are made exclusively on those identified modules. In this manner, the amount of time it takes to implement code changes in object-oriented applications is compressed from many months -- as is the case with cumbersome legacy applications that do not take advantage of object-oriented design -- to a couple days. This approach also reduces the number of potential errors introduced to a system with each code modification and release.

Another advantage of the distributed object framework relates to code distribution. With the majority of the distributed object application located on the server(s), distribution of code to the client is effectively handled by simply swapping the modified components on the server side, thus eliminating the need for massive updates to all clients. The next time a client accesses the server, the latest code releases will be provided automatically (and transparently), imposing no undo burden on the end-user.

6.1.4 Thin Clients

Since the major components of a distributed object application are located on a server (or a network of servers), the client-side application can be kept small and lightweight. This allows more of the clients' system resources to remain free while the bulk of the application processing is performed on the high-end servers.

With objects distributed across a network of machines, the thin client can access an infinite number of data repositories and legacy applications of unlimited size, thus making disk space and memory on the client side a non-issue.

And with respect to the initial distribution of an application, lightweight clients (Applets, Active-X objects, stand-alone applications) are small enough that the Internet is an ideal distribution mechanism.

6.2 Advantages and Disadvantages of RMI System

Java Remote Method Invocation (RMI) allows you to write distributed objects using Java. RMI provides a way for client and server applications to invoke methods across a distributed network of clients/servers running the Java Virtual Machine. Despite the fact that RMI is considered to be lightweight and less powerful then CORBA and DCOM, RMI still brings to the table some unique features, like distributed, automatic management of objects and the ability to pass objects themselves from machine to machine. We summarized the advantages and disadvantages below. Advantages to the existing systems are:

- Objects are passed by value.
- The server/ client can reconstitute the objects easily.
- Data type can be any Java objects.
- Any Java objects can be passed as arguments.
- Arguments has to implement the Serializable interface.

The main disadvantage is:

- Heterogeneous objects are not supported.

RMI is the simplest and fastest way to implement distributed object architecture due to its easy-to-use native-Java model. Therefore, it is a good choice for RAD prototypes and small-sized applications that are implemented completely in Java. The main issue with RMI is that it’s not as robust or as scalable as CORBA or DCOM solutions. For example, RMI uses a native-transport protocol, JRMP (which is not currently CORBA/IIOP compliant), and can only communicate with other Java RMI objects. This "single-language" crutch makes it impossible for RMI to interact

with objects not written in Java – like legacy applications – and prevents RMI from playing a more formidable role in large-scale enterprise solutions.

6.3 Future Work

The main infrastructure of the system has developed. The necessary components can be added to the system easily. This brings the usefulness of adapting this system to other systems. Although the system which is presented is implemented completely, some points are not finished yet. These are summarized below.

6.3.1 Access Control

Although access control is implemented in the system, it is very basic. It authenticates collaborator and does not allow the same collaborator in the system. Also this method helps the history mechanism in the system. This authentication can be enough for small network but be not enough for larger networks. A security mechanism must be set in order to protect password and user data.

Another missing feature is, controlling the collaborators' access level. At this time, all the users in the system have same rights. Users' rights and collaborative information can only set from the database administrator. This must be handled in the future. So the manageability of the system will increase.

6.3.2 Undo Mechanism

As stated before undo mechanism is complicated for graphical editing systems. A simple undo mechanism implemented. But redo and more than one undo are not supported by the system.

Undo mechanism is complex because it must handle access control and history mechanism. Also the addition of the redo mechanism makes the system more complex. Although undo mechanism is simple in the system, the more complex one can be built on this structure.

6.4 System Structure

The chat and whiteboard system is implemented using the JAVA structure RMI. A new distributed collaboration frame does not design. RMI is used for distribution

mechanism. RMI handles all network and registry work at the background. This permits easy programming interface and strong structure.

Some of the parts of the system are not defined fully. Especially the undo, history and authentication mechanism has some missing parts. But they can be added to system easily. The final structure of the system is shown in figure 3.17 below.

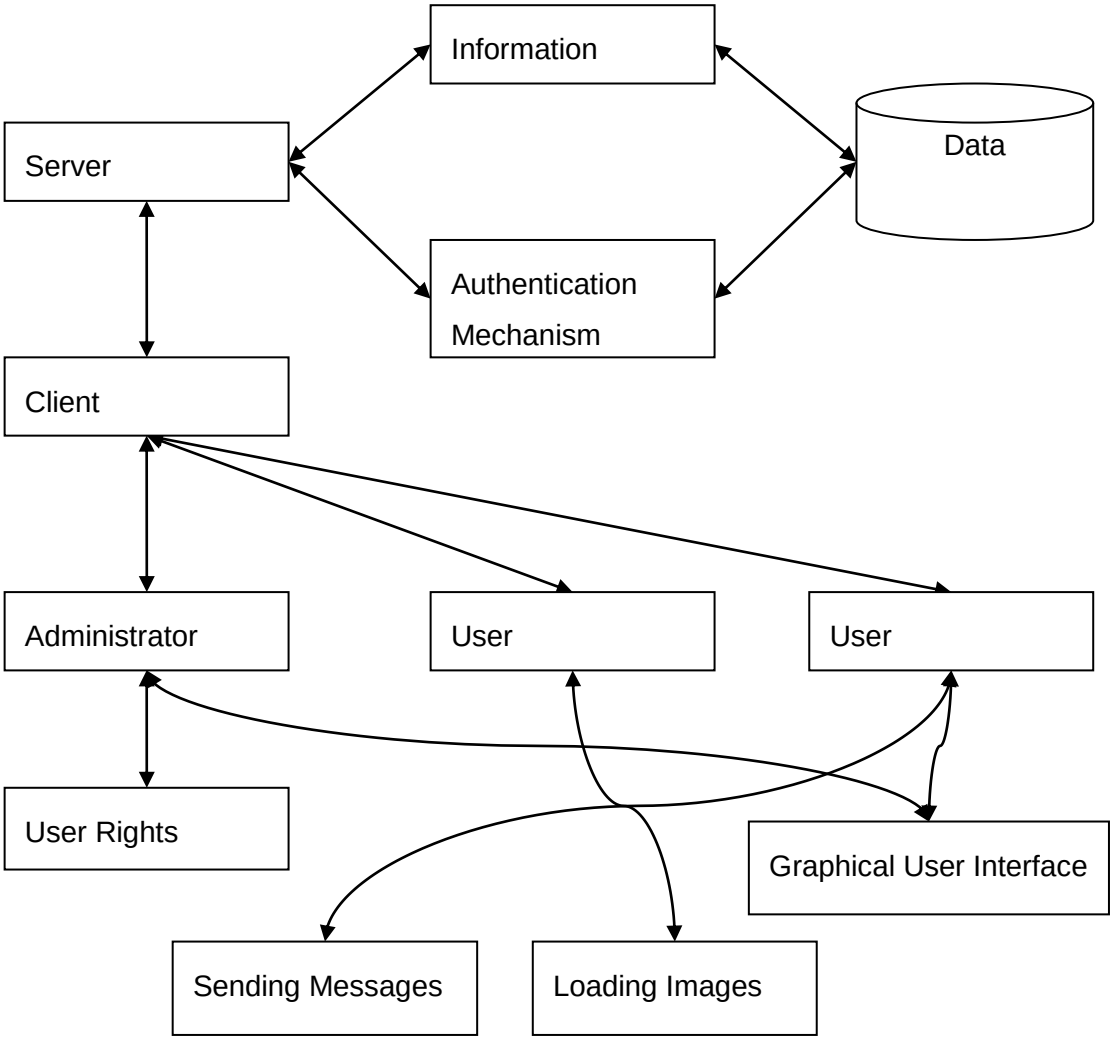

Figure 3.17 – The final system structure

The necessity of collaboration tools are increasing as the Internet usage increases. In the near future, many collaboration tools will be used, and will be developed. For faster and stronger development, distributed systems like CORBA or RMI systems should be used. These systems separate the network part of programming from the distributed part. Also reusability of systems will be increased with this approach.The system that is developed is a framework for collaboration tools. It can be used with its tolls or be improved.

REFERENCES

[1] **Pitt E. and Mcniff K.,** 2001. Java.rmi “The Remote Method Invocation Guide”. Pearson Education Limited, Great Britain.

[2] **Farley J.,** 1998. JAVA Distributed Computing. O’Reilly & Associates, Inc., United States of America.

[3] **Harold E. R.,** 1997. Java Networking Programming. O’Reilly & Associates, Inc., United States of America.

[4] **Wutka M.,** 2001. Using Java 2. Que, Indiana, United States of America.

[5] **Arnold A., Gosling C., and Holmes M.,** 2000. The Java Programming Language. Addison-Wesley, Great Britain.

[6] **Papaioannou T.,** 2000. On the Structuring of Distributed Systems: The Argument for Mobility, PhD Thesis, Loughborough University.

[7] **Janson F. and Zetterquist M.,** 2000. CORBA vs. DCOM, Master Thesis, Department of Teleinformatics, The Royal Institute of Technology.

[8] **B. Kvande,** 1996, The Java Collaborator Toolset, a Collaborator Platform for the Java Environment, Master Thesis, Old Dominion University.

[9] **N. Sharada, S. Gangishetty, K. Adusumilli, S. Hsu**, “ Application of Videoconferencing to Distance Education: A Case Study with the ClassPoint Software”, SCI’99, World Multi-Conference on Systems, Cybernetics and Informatics, Aug. 1-5, 1999, Orlando, Florida, pp. 122-130.

[10] **R. R. Raje, S. Mukhopadhyay, M. Boyles, N. Patel, A. Papiez, J. Mostafa**, “On Designing and Implementing A Collaborative System Using the Distributed-Object Model of Java-RMI”, WWW Journal, Special Issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents, In Press 1998.

[11] **H. Abel-Wahab, O. Kim**, “An Internet Collaborative Environment for Sharing Java Applications”, Information Technology Laboratory-NIST, IEEE-Future Trends in Distributed Computing, Oct. 1997, pp. 112-117.

[12] **H. Domel, J. Aceves**, “Floor Control for Multimedia Conferencing and Collaboration”, ACM Multimedia’97, 5(1), Jan. 1997.

[13] **Shen, H., and Deawan, P.**, “Access Control for Collaborative Environments,” Proceedings of the Conference on Computer-Supported Cooperative Work, Toronto, Ontario: ACM, 1992, 51-58.

[14] **Keith Edwards**, “Policies and Roles in Collaborative Applications”, ACM CSCW’96 Proceedings, Nov. 1996, Cambridge MA.

[15] **Craighill E., Lang R., Fong M., Skinner K.,** “ CECED: A system for informal multimedia collaboration”, Proceeding of ACM Multimedia 1993, Anaheim, Calif., New York, NY, ACM Press, pp 437-445.

[16] **Garcia-Luna-Aceves JJ., Craighill EJ., Lang R.,** “Floor management and control for multimedia computer conferencing”, Proceedings of the 2nd IEEE Comsoc International Communications Workshop, 1989, Ottawa, Canada.

[17] **Rajan S., Rangan PV., Vin HM.,** “A formal basis for structured multimedia collaboration”, Proceedings of the 2nd IEEE Multimedia Computer and System Conference 1995, Washington, D.C., IEEE Computer Soc. Press, Los Alamitos, CA, pp 194-201.

[18] **Yavatkar R., Lakshman K.,** “Communication support for distributed collaborative applications”, Multimedia Systems 2 1994, pp 74-88.

[19] **S. Hirano, Y. Yasu, and H. Igarashi,** “Performance Evaluation of Popular Distributed Object Technologies for Java”, Proceeding of the ACM 1998 Workshop on Java for High Performance Network Computing, Stanford, Palo Alto, Calif., Feb. 1998.

[20] **C. A. Ellis and S. J. Gibbs**, “Concurrency Control in Groupware Systems”, ACM SIGMOD “89 Proceedings, Portland, Oregon, 1989.

[21] **Micheal Boyles, Rajeev Raje, and Shiaofen Fang,** “CEV: Collaborative Environment for Visualization Using Java RMI”, ACM Workshop on Java for High Performance Network Computing, ACM Press, 1998.

[22] **A. Baratloo, M. Karaul, H. Karl, and Z. Kedem**, “Knittingfactory: An Infrastructure for distributing web applications”, Technical Report TR 1997-748, New York University, November 1997.

[23] **Lukasz Beca, Gang Cheng, Geoffrey C. Fox, Tomasz Jurga, Konrad Olszewski, Marek Porgony, Piotry Sokolowksi, Tomasz Stachowiak, and Kryzystof Walczak.** , “Tango – a Collaborative environment for the world wide web”, Technical Report, Northeast Parallel Architectures Center, Syracuse University.

[24] **RFC 1831:** Srinivasan, R., Remote Procedure Call Method Protocol, 1995.

[25] **Java Remote Method Invocation Specification,** Revision 1.7, Java 2 SDK, Standard edition, v1.3.0, 1999, distributed with Java JDK 1.3, or via the Java Software home page.

[26] **The Java Software home page** is at http://java.sun.com/. The Java RMI home page is at http://java.sun.com/products/rmi/. Links on these pages includes the RMI White Paper, the RMI specification, online documentation, a note on RMI and SSL, the RMI-USERS mailing list, online tutorials.

[27] **Object Management Group specifications (CORBA, IIOP);** at http://www.omg.org/ .

[28] **MSDN Library;** at http://msdn.microsoft.com/.

CURRICULUM VITAE

A. Çağatay TUNALI was born in 1976 in İstanbul. He attended Marmara University, Engineering Faculty, Computer Engineering Department at 1995. He graduated Computer Engineering Department with a fourth degree as an honor student at 2000. Since 2000, with partnership of Garanti Technology and İstanbul Technical University; he performs assistant work in Computer Engineering Department of İstanbul Technical University.